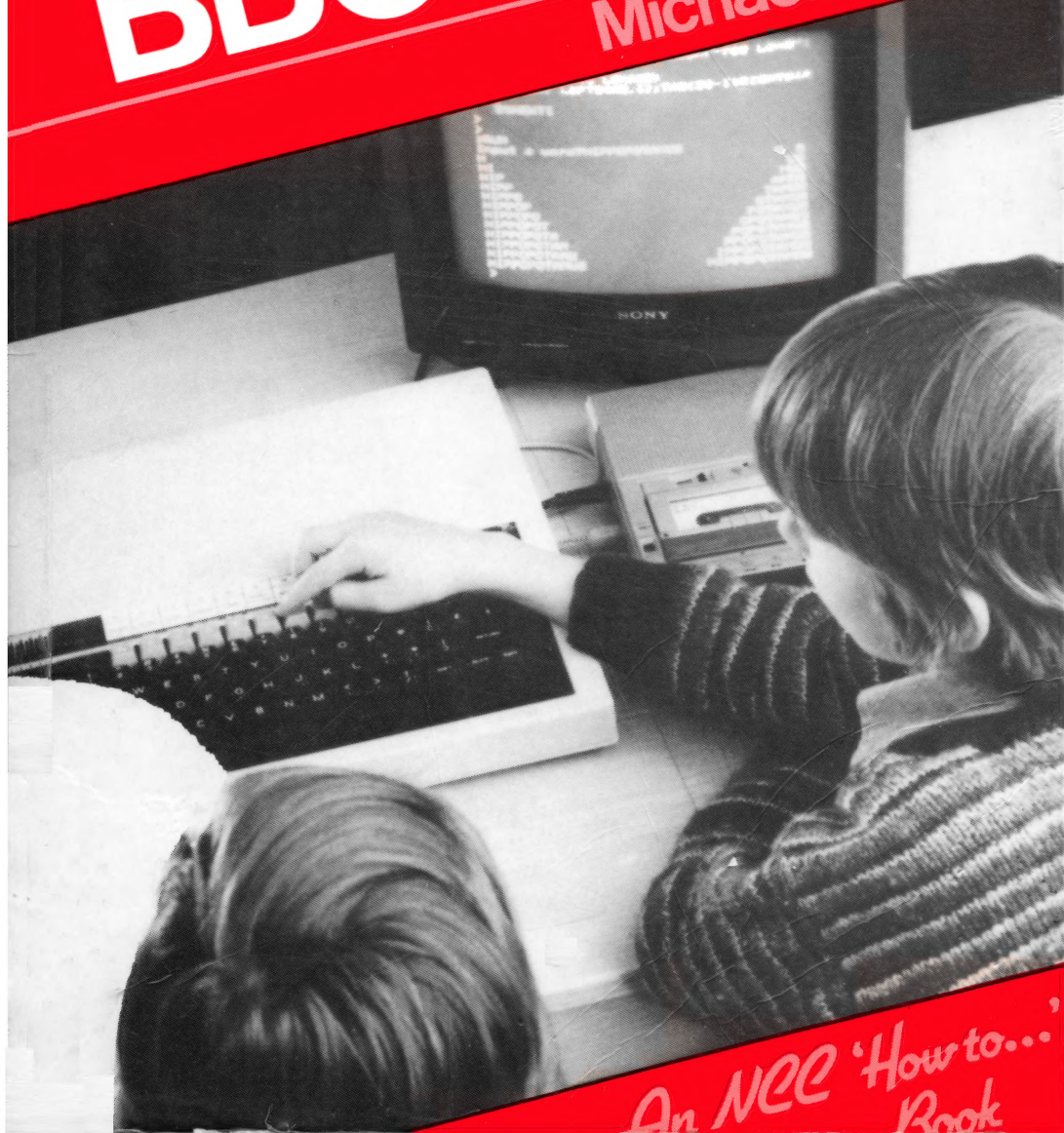


A YOUNG PERSON'S GUIDE TO **BBC BASIC**

Michael Milan



An NCC 'How to...' Book

A Young Person's Guide to BBC BASIC

Michael Milan

PUBLISHED BY NCC PUBLICATIONS

British Library Cataloguing in Publication Data

Milan, Michael

A young person's guide to BBC BASIC.

1. BBC Microcomputer – Programming
2. Basic (Computer program language)

I. Title

001.64'24 QA76.8.B3

ISBN 0-85012-393-3

© THE NATIONAL COMPUTING CENTRE LIMITED, 1983

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, without the prior permission of The National Computing Centre Limited.

First published in 1983 by:

NCC Publications, The National Computing Centre Limited,
Oxford Road, Manchester M1 7ED, England.

Typeset in 11pt Times Roman and printed in England by
UPS Blackburn Limited, 76-80 Northgate, Blackburn,
Lancashire.

ISBN 0-85012-393-3

Acknowledgements

Of the many people who helped with this book I would especially like to thank the following:

My wife, Corinne, who became a computer widow while it was written;

My two sons, Stephen (who wrote many of the programs) and Christopher (who helped test them);

Mike Turner, for his super drawings;

Tony Squires, for some good ideas;

Finally, but not least, Anne, Eunice, Helen, Karen and Margaret for word processing, with saintly patience, the many versions of the manuscript.

Contents

	<i>Page</i>
Acknowledgements	
1 What's It All About?	9
What is a Computer?	10
How to Connect Up and Switch On	13
2 Making a Start	15
Editing the Program	19
Loading and Saving Programs	25
3 Counting on Your Computer	29
4 Branch Line	37
5 Looping the Loop	47
RND	53
6 Hip Hip Array	55
7 How To Proceed	63
8 Keeping Tabs on Things	75
9 Design for Good Programming	89

10 A La Mode	93
11 A Colourful Character	107
TELETEXT Colours	107
Colours in MODES 6 to 0	116
12 Play It Again, Computer	127
Channel	128
Amplitude	128
Pitch	128
Duration	129
13 It's All a Plot	139
Appendix 1	
Debugging	159
Syntax Errors	159
Logical Errors	162
Index	163

1 What's It All About?

Most microcomputers look similar from the outside. Usually they are in a flat case, with a keyboard, rather like a typewriter. Then there is usually a television set, so that you can see what has been typed; and a socket so that a cassette recorder can record the programs. This way programs can be stored and played back into the computer without a lot of typing in.

Although most micros look similar, different types are made to work in slightly different ways and so this book has been written to go with the BBC and Electron computers made by Acorn, and also the Acorn Atom with a BBC BASIC ROM.

It is assumed that you have the use of one of these machines and can work through the book by trying out all the activities shown. To follow the book you need nothing more than a standard BBC Model A (or the Model B which has more memory and other facilities which are beyond the scope of this book). The Electron and the Atom use the same language, but some of their facilities are different.

You will also need a cassette recorder and a black and white TV. Of course, in order to see the results of Chapter 11, you will need the use of a colour TV; but for learning to program, a black and white set will do.

A good way to begin to understand computer programming is to type in programs from magazines. There are many magazines published each month aimed at the personal computer market. For young readers, *Your Computer* has articles, competitions and

programs which will be of interest. Not all the programs will be written specially for the Electron or the BBC, but with practice you will be able to adapt programs written for other computers and, of course, get ideas for programs of your own. Another magazine that will be of interest to BBC microcomputer owners is *Acorn User*. This covers the BBC microcomputer, the Atom and the Electron.

For real enthusiasts at writing their own programs for the BBC computer, BEEBUG (the BBC Computer Independent User Group) issues a magazine (ten times a year) packed with programming advice.

Of course there are lots of other magazines, weekly and monthly, so look around and see which ones have articles and programs that interest you most.

Don't spend a lot of money buying programs already on cassette, you'll just be tempted to put them into the computer and play with the games. That way you won't find out how they work. For the price of one program cassette you can buy several magazines containing lots of programs and articles on programming techniques.

WHAT IS A COMPUTER?

All computers are machines that obey a **program** of instructions. Although the computer may seem to do amazingly clever things, it is just a machine, and a pretty stupid one at that. It can *only* perform instructions that have been written in a way it can understand. The whole skill of programming is to be able to give the computer instructions *it* understands, and that will make it do what *you* want it to. To do this we have to use a language that the computer recognises. This book will help you learn the computer language called BASIC, and, more especially, the version of BASIC that has come to be known as BBC BASIC.

BASIC has many words in it that are very like English. You will certainly be able to recognise at least 2000 words in English, but your computer can only recognise about 120 words of BASIC. So you can see that compared to learning Welsh or German or French, learning BASIC should be quite easy, because there are so few words. The snag is that, because the computer is stupid, if you

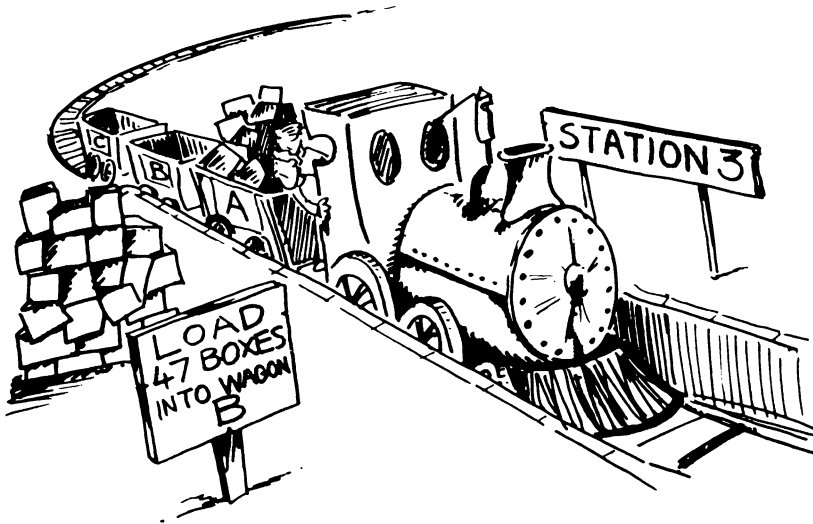


make the slightest mistake the computer will not be able to understand what you are trying to say (whereas the Welsh, Germans and French will try and help you along). Still, if you make a mistake, you will get a message on the screen to help you trace where you went wrong in the program.

We've already said that a computer is a machine that follows instructions. Think of it as a goods train, and the program is the track with stations on it. The train has several wagons to hold goods, and each station has a notice board with instructions.

The train must start at the end of the line and at each station must stop and load or unload goods according to the instructions. A series of instructions could look like this:

- 1 START
- 2 LOAD 24 BOXES INTO WAGON A
- 3 LOAD 47 BOXES INTO WAGON B
- 4 TAKE WAGON A'S BOXES AND WAGON B'S BOXES AND LOAD THEM INTO WAGON C
- 5 DISPLAY CONTENTS OF WAGON C
- 6 STOP



The train can only follow the track, it can't go to one side or the other and so all the instructions are followed.

This is a very simple program (and not written in BASIC). Programs can be much longer and include branches (like railway points) so that the program can follow different routes when running.

So, switch on your computer and start at the beginning of the next chapter and try out all the activities. Don't be afraid to experiment – that's the best way to learn, and don't worry, you can't break the computer just by typing in the wrong thing. The activities that are part of the chapters are written in an order which will introduce you gradually to new BASIC commands, so it's best to start at the beginning and work through. All the programs work on a Model A BBC computer with 16K of memory (although there are suggestions how some will work better with 32K). Most of the programs will also work with an Acorn Electron or an Acorn Atom with a BBC BASIC ROM.

Have fun!

HOW TO CONNECT UP AND SWITCH ON

The BBC computer comes complete with an attached mains lead and moulded 13 amp plug. This plug should *not* be removed. If the electrical sockets in your home do not fit this type of plug then consult a qualified electrician.

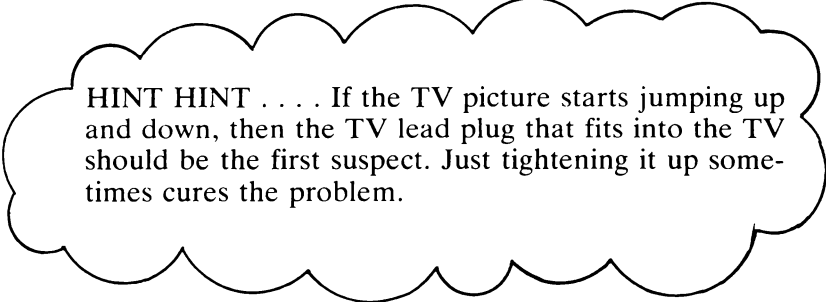
Two other leads are supplied, a TV lead with a plug at each end, and a cassette recorder lead with a 7 pin DIN plug at one end and something else at the other. (See **LOADING AND SAVING**, in Chapter 2, about this lead.)

One of the plugs on the TV lead has a long central pin sticking out from the body of the plug. This is a PHONO plug and goes into the socket at the left-hand end of the back panel of the computer (marked "UHF out"). The other end plugs into the aerial socket of your TV set.

Plug in both computer and TV set and switch them on. The computer should give off a double "beep" sound, and the TV picture will look like a snow storm. Turn the volume control on the TV right down. You don't need the sound and it will make tuning-in much more peaceful!

If you have a rotary tuner on the TV, then turn it to around channel 36 and you should get a black screen with "BBC COMPUTER" printed in the top left-hand corner.

If you have a push-button tuner, then press a button that you don't normally use and tune that channel to the computer.



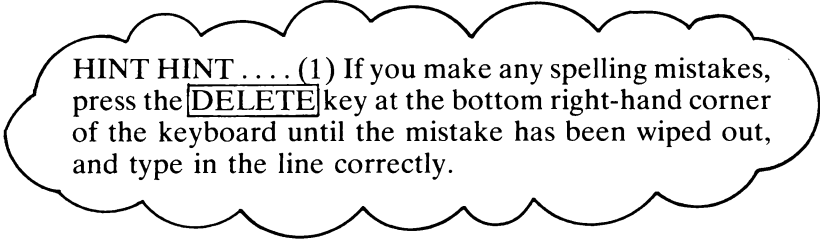
HINT HINT If the TV picture starts jumping up and down, then the TV lead plug that fits into the TV should be the first suspect. Just tightening it up sometimes cures the problem.

2 Making a Start

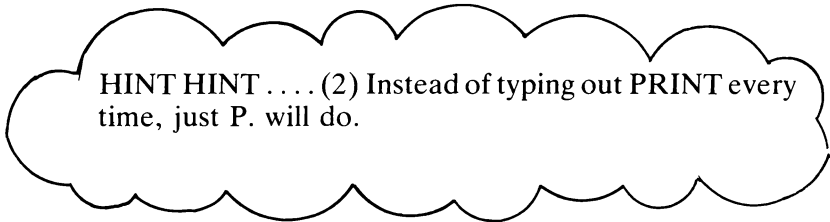
In this chapter we are going to start programming using the keyboard. The first programs will be very simple, but by trying the experiments you will soon get the idea of how to enter programs and alter them. One of the best things about computer programming is that you can always think of ways to improve your programs.

In the last chapter we saw that a computer program is a series of instructions that a computer can understand and follow. In this first program we are going to ask it to print a name on the screen. Before you start, press the CAPS LOCK key (so that the central red light at the bottom left of the keyboard is lit). Type in the following lines just as they are shown here, remembering to press the RETURN key at the end of each line.

```
10 PRINT "WHAT IS YOUR NAME" RETURN
20 INPUT NAMES$ RETURN
30 PRINT "HELLO" RETURN
40 PRINT NAMES$ RETURN
50 END RETURN
```



HINT HINT . . . (1) If you make any spelling mistakes, press the **DELETE** key at the bottom right-hand corner of the keyboard until the mistake has been wiped out, and type in the line correctly.



HINT HINT . . . (2) Instead of typing out PRINT every time, just P. will do.

When you have typed in all the program, type RUN
`RETURN`

If all goes well you should see:

```
RUN
WHAT IS YOUR NAME
?
```

which asks you to type in your name, so here goes:

```
CHRISTOPHER RETURN
(or whatever your name is)
```

and the computer prints:

```
HELLO
CHRISTOPHER
```

Already you have typed in your first computer program. If the program didn't run for you or you got:

```
MISTAKE
or SYNTAX ERROR
```

then you've got a **bug** in the program. Type LIST and `RETURN` and check the spelling carefully with the list above. If any lines are wrong then just type those lines again (including the line number) at the bottom of the screen, remembering to press the `RETURN` key after each one. Another important thing to check is that all BASIC commands are in CAPITAL LETTERS. If not, the computer simply won't recognise them. There is a check-list for **debugging** programs in Appendix 1 at the end of this book.

Now let's see how the program worked. Type LIST and

RETURN and you will see:

```
10 PRINT "WHAT IS YOUR NAME"  
20 INPUT NAMES$  
30 PRINT "HELLO"  
40 PRINT NAMES$  
50 END
```

We gave the computer two quite different types of instruction. RUN and LIST are called **DIRECT COMMANDS**. These instructions are obeyed straight away (if there is a stored program to be run or listed). The program lines (those numbered 10, 20, etc) are called **STATEMENT LINES**.



Direct Commands

PRINT, INPUT and END are statements and are some of the special BASIC words that the computer can recognise. The statement lines are numbered and the computer acts on them in the correct order. That's a very good thing because you can type more lines in later, and the computer will act on them as if they had been typed in the right order. That's why we've numbered the lines 10, 20, 30 and so on. If we want to add extra lines later we can use numbers in between. Let's try that out.

Type in:

```
25 PRINT RETURN
```

Now type LIST **RETURN** and you will see that line 25 is in the right place.

Now we'll look at each line in detail and see what each does.

10 PRINT "WHAT IS YOUR NAME"

When the computer comes to the statement PRINT in a program it prints onto the screen any word or STRING of characters that are on that line between **quotation marks** (also known as **inverted commas** – "). So when the computer comes to line 10 it will print WHAT IS YOUR NAME onto the screen.

20 INPUT NAME\$

When it comes to an INPUT statement the computer will print a **question mark** (?) on the screen and waits for you to type something in at the keyboard. Before it can continue with the program you must press the RETURN key.

NAME\$ is a **variable**. We store any words, letters or numbers that we wish the computer to handle in its memory. These are called variables. The computer's memory is like a lot of different railway wagons. We can tell the computer to store words, letters or numbers in those wagons, and they will stay there until we do something to alter them. So that the computer can find them again when we want them, we tell the computer what name to call the wagon. This time we have called the wagon NAME\$, but we could have called it after any single letter of the alphabet, from A\$ to Z\$. We could have used two letters together, such as AA\$, AB\$ right through to ZZ\$, or else we could use a complete word (like NAME\$ or even name\$). We must be careful, though, that we don't use one of the special BASIC words – called RESERVED words – that the computer recognises as STATEMENTS or COMMANDS. There is a list of these in the User Guide (which came with the computer).

It's very useful to be able to use **whole words** for variable names. They may take a bit longer to type in (and you have to be sure that they are correct every time that you use them in the program), but they make a program much easier for *you* to read when you want to alter or improve it.

Using different **variable names** we can have a lot of different variables in the computer's memory at one time without mixing

them up. You will run out of memory space long before you run out of variable names!

The variable that we have used so far has got a \$ (dollar) sign at the end. This tells the computer to expect a STRING to be input. That is a letter, or a line of letters, or even letters and numbers. You can easily store up to 256 letters and numbers (known as CHARACTERS) in any one string. These string variables are very useful in computing because a lot of computer work is to do with storing and printing words. But if you want to store numbers, so that you can do arithmetic, then you will need to use a different sort of variable as we shall see in the next chapter.

25 PRINT

If a PRINT statement isn't followed by anything, then the computer will just leave a blank line on the screen. RUN the program to see this and then LIST again.

30 PRINT "HELLO"

This line tells the computer to print out HELLO because that is what is between the quotation marks.

40 PRINT NAME\$

This line tells the computer to print out whatever is in the variable store that we have called NAME\$. In line 20 we told the computer to store what was INPUT into NAME\$. So whatever you typed in then will be printed out again now.

50 END

It shouldn't be too hard to guess what this line does, it tells the computer that this is the end of the program. This line isn't even necessary in this program because if the computer can't find any more lines, it knows that the end has come. But in some of the programs that we will be writing later on, the program may end somewhere in the middle of the lines so we will need to use END then.

EDITING THE PROGRAM

You've typed a program into the computer (computer people would call this KEYING the program IN). You've RUN the

program and you've seen how you can add an extra line at any time. Now let's try changing the program. You should have the program LISTed on the screen. If not, type LIST and `RETURN`

Now type

25 `RETURN`

and then LIST again.

You will see that the computer has now forgotten line 25.

Getting rid of lines is very easy, but it's also easy to change lines that you want to stay. One way is simply to retype the whole line. That's what you just did when you typed 25. That replaced the line 25 that was there, with a blank line.

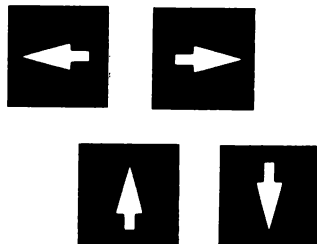
Try typing this:

30 PRINT "GOOD MORNING" `RETURN`

and then type LIST `RETURN`

Now the new line 30 is in the place of the old one.

There are ways of editing which are even easier than this. To be able to do this, you must look for the **CURSOR CONTROL KEYS**. The cursor is the little flashing underline on the screen that tells you where the computer is going to print the next character. The cursor control keys are the four keys at the top right hand corner of the keyboard that look like this:



LIST the program so that it is in a convenient part of the screen. Now press the `↑` key and you will see that the cursor moves up the screen, keep pressing the key and it will carry on moving. Then try the `→` , `↓` and `←` keys as well.

Now using these keys, move the cursor until it is at the beginning of line 10. Then press the key marked **COPY** in the bottom right-hand corner of the keyboard.

Now you will see that each time that you press the copy key, a character from line 10 is copied onto the bottom line of the screen. Carry on until the whole of line 10 has been copied and then add the character ; using the keyboard. Now press **RETURN**. To see what has happened type LIST **RETURN**.

If all has gone well, line 10 will now read:

```
10 PRINT "WHAT IS YOUR NAME";
```

Now type RUN **RETURN**

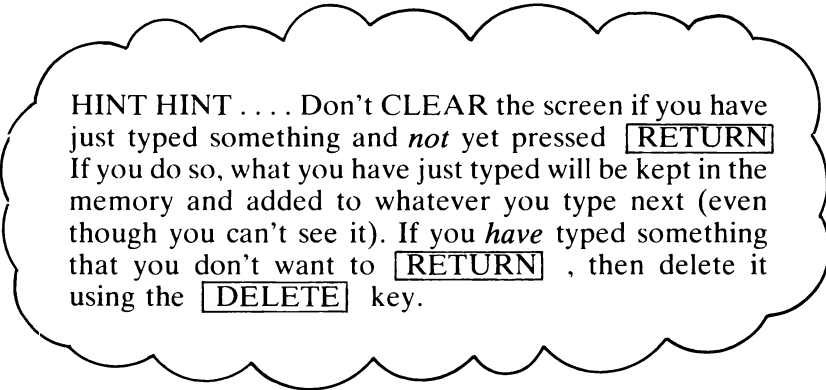
Do you see any change in the way that the program runs? The question mark for the INPUT comes on the same line as "WHAT IS YOUR NAME". Using the character ; we have told the computer *not* to move to a new line after PRINTing. Finish running the program and then LIST again. Now try COPYing line 30 to add a semicolon ; after "GOOD MORNING". (Remember to add it after the quotation mark, or else it will just be printed as "GOOD MORNING;".) Then RUN the program again.

Now you will see that there is a slight problem because there isn't a space between "GOOD MORNING" and the name that you entered. This is because when there is a semicolon, the computer prints the variable (NAME\$) immediately after the end of the string ("GOOD MORNING"). So now we need to insert a space in the right place using the COPY key. Using the cursor control key, move the cursor up to line 30 (LIST the program again if it isn't visible on the screen). Then COPY line 30 as far as the G in MORNING. Then using the space bar insert one space between the G and the ", then copy the rest of the line and press **RETURN**. RUN the program again.

Sometimes it's useful to be able to clear the screen. To do this, look for the **CONTROL** key (marked CTRL). It's on the left-hand side of the keyboard (next to the A key). Press **CONTROL** and keeping it down, press L at the same time.

When you do this the screen will clear and the cursor goes to the top left-hand corner. But don't worry, the program is still in the

memory. Type LIST RETURN to check this.

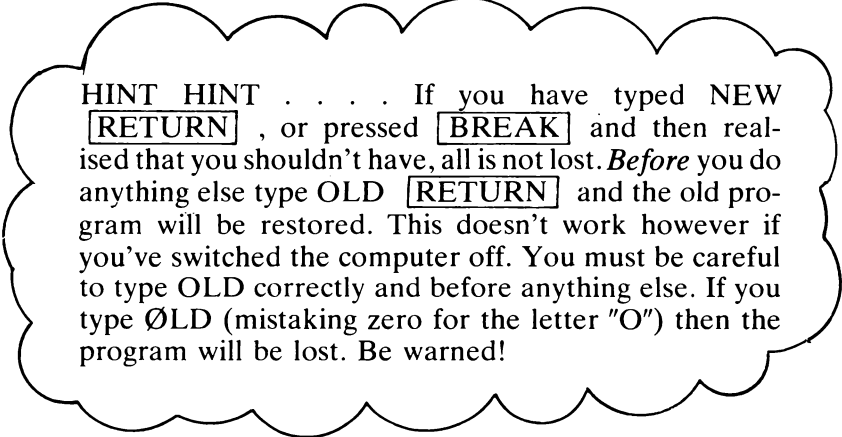


HINT HINT . . . Don't CLEAR the screen if you have just typed something and *not* yet pressed RETURN. If you do so, what you have just typed will be kept in the memory and added to whatever you type next (even though you can't see it). If you *have* typed something that you don't want to RETURN, then delete it using the DELETE key.

Well, it wasn't a very big program, but already you have found out how to tell the computer to do all the following things:

RUN
LIST
PRINT
INPUT
INSERT NEW LINES
DELETE LINES
INSERT NEW CHARACTERS
DELETE CHARACTERS
MOVE CURSOR AROUND SCREEN
CLEAR SCREEN

Now type NEW RETURN and this will clear the computer's memory for the next program. Another way to do this is to press the BREAK key in the top right-hand corner of the keyboard.



HINT HINT If you have typed NEW **RETURN** , or pressed **BREAK** and then realised that you shouldn't have, all is not lost. *Before* you do anything else type OLD **RETURN** and the old program will be restored. This doesn't work however if you've switched the computer off. You must be careful to type OLD correctly and before anything else. If you type ØLD (mistaking zero for the letter "O") then the program will be lost. Be warned!

Here's another program to type in. Make sure that you put in the spaces where they are shown.

```
10  A$="TYPE IN"
20  B$=" A"
30  C$="NOTHER"
40  D$=" NAME"
50  PRINT A$;B$;D$
60  INPUT E$
70  PRINT A$;B$;C$;D$
80  INPUT F$
90  PRINT D$; " NO. 1 WAS "; E$
100 PRINT D$; " NO. 2 WAS "; F$
110 END
```

Can you try to work out what this program will do before you RUN it? It has one new idea. In lines 10, 20, 30 and 40 the program gives the computer strings to store in different memory locations. Here the = (equals) sign is used to mean "**is given the value of**". So line 10 means A\$ is given the value of "TYPE IN". Then in lines 50

and 70 it is told to PRINT the variables out again in different ways. You'll see how important it is to put the spaces in the correct places (especially if you got them wrong!).

RUN the program to see how well it worked. Now *you* try to write a program that makes up two different sentences from the same words. If you can't think of anything else, try these words:

WHAT IS MY COMPUTER CALLED?

MY COMPUTER IS CALLED ARNOLD

(Don't forget to `BREAK` before you type in a new program or else things can get very mixed up!)

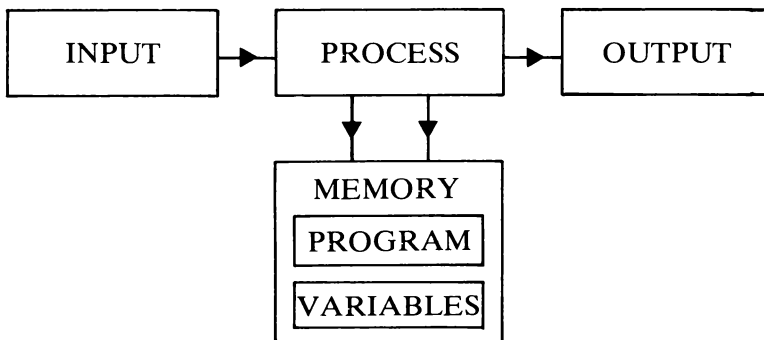
So now we can see the way in which any computer operates.

There is **input** (in our case from the keyboard)

There is **output** (in our case to the screen)

There is **process** (the microcomputer)

There is **memory** (which holds both the **program** and the **variables**)



LOADING AND SAVING PROGRAMS

How to play program cassettes into your computer and how to record your programs on cassette

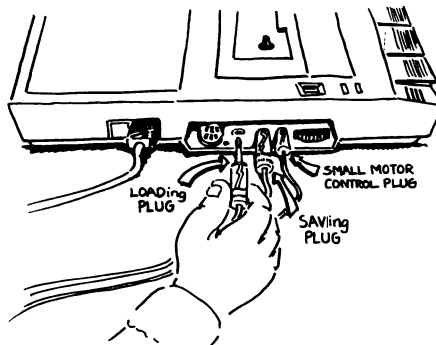
If you have bought your BBC computer from an Acorn dealer, then he will have supplied a lead to suit your cassette recorder. If you bought your computer by mail order, then you were possibly supplied with a lead with bare wires at one end. This will need making up with plugs. If you are not sure how to do this then you should get someone who knows about electronics to help you, or get a properly made up lead from one of the companies who advertise in computer magazines.

If you don't already have a suitable cassette recorder then you should choose one with care, because experience has shown that not all are really suitable. It is not necessary to have a very expensive recorder, but it seems that in general those that use DIN plugs are easier to use. There should also be a small jack plug to switch the motor on or off.

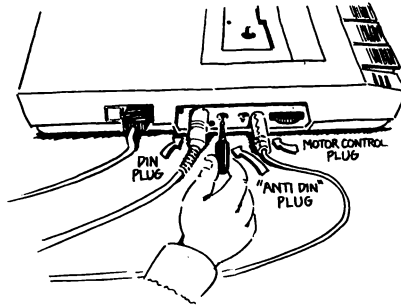
We had a lot of difficulty with SAVING and LOADING programs until we bought a "computer-compatible" recorder from a high street bookshop. This has proved most satisfactory. If you already have a cassette recorder, of course, there is no reason to buy another one unless you have a problem with the one that you have.

Connecting

1. With jack plugs



2. With a DIN plug



Many cassette recorders make a loud noise when **LOADing using the **DIN** plug. Put a blank 3.5mm jack plug into the “earphone” socket and you will be able to **LOAD** in peace.**

Loading

Your first go at **LOADing** will probably be with the **WELCOME** cassette supplied with the computer. This is useful because the first program on that tape helps you to discover the best **VOLUME** setting on your recorder.

There are two commands that you can use: **LOAD** and **CHAIN**. Both **LOAD** and **CHAIN** will search for named programs. But probably it is most useful to use the command:

LOAD " " RETURN

Press **PLAY** on the cassette recorder and if the volume is correctly set, the name of the next program on the cassette is soon shown on the screen. Then the screen shows a series of numbers, which are the numbers of the blocks in which the program was recorded. The numbers are in **hexadecimal**, so that after 09 there is 0A, 0B, etc, to 0F and then 10. When all the program has been loaded, another number with four digits is shown and the computer gives off a beep. Don't stop the recorder before the beep. Then you can **RUN** or **LIST** the program.

CHAIN " " RETURN

This is the same as **LOAD** " " except that as soon as the program is **LOADED** it is automatically **RUN**.

Problems

If the program doesn't LOAD properly this will show itself in one of three ways:

- 1 **Nothing happens on the screen.** Check connections and VOLUME setting. Also, if there is a TONE control on the recorder this should be set to 10 or MAX or HIGH.
- 2 **The text on the screen scrolls up with the word Block? or Data?** It is only necessary to rewind the tape for the blocks actually missed. Try a fine adjustment of the volume control. If you use jack plugs it may be necessary to unplug the RECORD lead when LOADING, and the REPLAY lead when SAVING.
- 3 **The program appears to LOAD correctly, but there is no beep at the end, and the message "Bad Program" is displayed when it is RUN.** Try LOADING again and watch carefully. The first block to be shown on the screen should be 00. If the first block shown is 01, then the first block has been **corrupted** and cannot be read. This has been a problem which happens occasionally with early BBC computers, but shouldn't happen with later ones. If you have this problem then a solution is given under Saving Programs, below.

Saving Programs

If you have a program in the computer's memory that you want to record, then you'll need to think of a name for it. You can call it anything that helps you remember what the program is about, provided that the name has no more than eight letters. Let's say we are going to call our program "SORTLIST".

To SAVE this on to cassette, you will need a cassette on which to record. Short cassettes are better than long cassettes, because with long cassettes it is either very difficult to find the program that you want, or you waste a lot of tape. C10 to C15 cassettes are sold for computer use and these are very convenient.

Put the cassette into the recorder and type:

SAVE "SORTLIST" RETURN.

The computer will reply with

"RECORD AND RETURN".

Start the cassette recorder. Make sure that the tape is past the leader and then press **RETURN** . The computer will then start sending the program to the cassette recorder and the screen will show the blocks as they are SAVED (in the same way as when programs are LOADED).

Keep a note of what programs you record on the cassette, including the counter number (always start the counter at 000 at the start of the cassette). Many programs have been tragically lost because something else has been recorded over them! If you over-record even a small part of a program it can be ruined.

Many people like to SAVE a program twice, in case there is a problem with the recording. If you want to be sure, you can check the recording before you clear the program from the memory. Here is one way to do it:

SAVE the program normally.

Rewind the cassette to the beginning of the program.

Type in:

*LOAD 8000 **RETURN**

The program is then LOADED, but into a part of the computer where it can do no harm to the original program which is still in the memory.

If the recorded program appears to LOAD correctly, then the recording is all right. If it won't LOAD, then SAVE the program again and test again.

3 Counting on Your Computer

Your computer can, of course, count. We know what you're asking yourself. "Can my computer do my homework for me?". Well, the answer is certainly "yes . . . but". Remember that the computer can only do exactly what it is told to. The great strength of a computer is that once it is told how to do something, it can do it very quickly and very accurately. Even so, it can't work out how to solve problems by itself.

If you want your computer to do your homework for you, you have to work out exactly how to do it, and then tell your computer how, in a way that it can understand.

Doing arithmetic on your computer is easy. Try typing this:

P. 2+2 RETURN

and the computer will print:

4

We have used a DIRECT COMMAND just as we did in the last chapter. Here are some more:

P. 8-3 RETURN

P. 6/2 RETURN (this means $\frac{6}{2}$ or 6 divided by 2)

P. 7*3 RETURN (this means 7×3)

Try some more for yourself.

Now we can try these functions in a program.

Type this in:

```

10 A$="INPUT A"
20 B$="NOTHER"
30 C$=" NUMBER"
40 PRINT A$;C$
50 INPUT A
60 PRINT A$;B$;C$
70 INPUT B
80 PRINT A;" PLUS ";B;" = ";A+B
90 PRINT A;" MINUS ";B;" = ";A-B
100 PRINT A;" DIVIDED BY ";B;" = ";A/B
110 PRINT A;" TIMES ";B;" = ";A*B
120 END

```

RUN the program a few times INPUTting different numbers each time. Now type LIST RETURN .

Have you noticed that as well as the string variables A\$,B\$ and C\$, there are also A and B (starting in lines 50 and 70). Variables without a \$ sign just stand for numbers, not words or strings. You can use A and A\$ in the same program, the computer will not mix them up.

The computer can also do several things in one sum – for example, try this (don't BREAK yet):

P. $2+3*4$

(before you press RETURN try to work out what the answer will be – will it be 20 or 14?)

When doing a sum with several different operations, your computer will do them in a special order. It will always do * (times) and / (divide) before + (plus) or – (minus).

Try to work out the answer to these yourself before you try them.

P. $15-2*5$

P. $60/4+5$

P. $5+16*3-4$

P. $20/5+4*2$

It's important to know what answer to expect, because when we want to program the computer to do something, we must know how to tell it exactly what to do.

If we want it to do things in a different order we can tell it how to do so using brackets (). Your computer will do anything inside brackets first.

Try:

$$P. 15-2*5$$

and then:

$$P. (15-2)*5$$

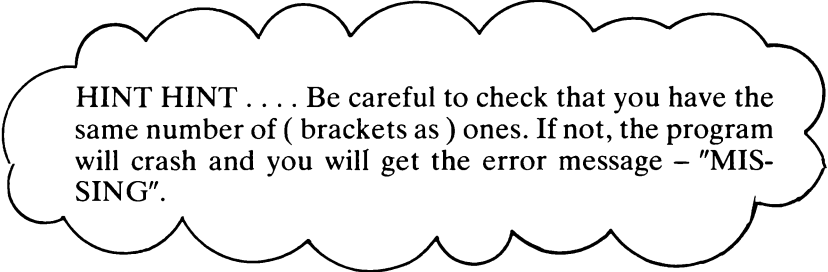
Try to work these out yourself first, before trying them on your computer.

$$P. 6+15-3*2$$

$$P. 6+(15-3)*2$$

$$P. (6+15-3)*2$$

$$P. (6+15)-3*2$$



HINT HINT . . . Be careful to check that you have the same number of (brackets as) ones. If not, the program will crash and you will get the error message – "MIS-SING".

Your computer can also do other types of sums which would be complicated for us to do by ourselves. Examples of these are **squares** and **square roots**. If you **square** a number, you multiply it by itself. Thus the square of 5 (or 5 **squared** as it is usually called) is $5*5 = 25$. 10 squared is $10*10 = 100$.

Try this:

$$P. 5*5$$

5 squared is usually written down like this: **5²**.

So try this:

P. $5 \wedge 2$

(this may appear as $5 \uparrow 2$ on the screen. Do you see that this is like 5^2 ?).

Here we are **raising 5 to the power of 2** (this is a mathematical way of saying that we multiply it by itself).

Now try this:

P. $5 \wedge 3$

The answer will be 125 which is $5 * 5 * 5$, or 5 raised to the power of 3.

Coming back to **squares**, the computer can also work out the **square root** of a number. If you start off with a number, let's say 25, then the **square root** is the number which would have to be **squared** to make 25. In this case the answer is simple (because we know it already). 5 is the **square root** of 25. But the computer can work out the square root of any number.

Try:

P. SQR(25)

P. SQR(100)

P. SQR(2)

P. SQR(3*5)

P. SQR(3.1412)

Now we can try putting this into a program. If you haven't typed NEW yet, then alter the last program as follows:

```

80 P. "THE SQUARE ROOT OF ";A;" TIMES ";B;
   " = ";SQR(A*B)
90 

|        |
|--------|
| RETURN |
|--------|


100 

|        |
|--------|
| RETURN |
|--------|


110 

|        |
|--------|
| RETURN |
|--------|


```

Try running this with different numbers. Try entering the same

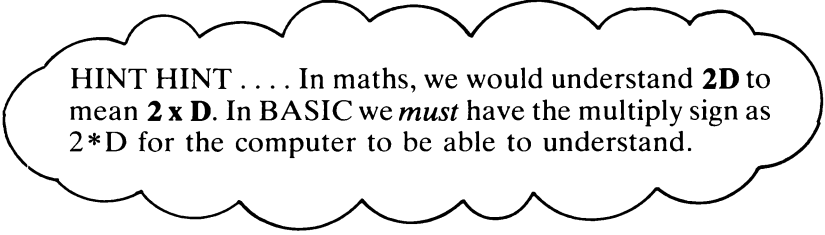
numbers twice in the same RUN.

Now **BREAK** and try writing a program which will INPUT four numbers and then find their average. To do this the computer must add them together and then divide the total by four. When you have written it, and entered it into the computer, test it by INPUTting the numbers 5, 7, 14 and 6. If the program is correct the answer will be 8. Then try running it with other numbers.

Here's a program to work out the area of a rectangle. It invites you to input the lengths of the sides. (Don't forget to **BREAK** or type NEW before you enter it.)

```
10 INPUT "HOW WIDE",WIDE
20 INPUT "HOW DEEP",DEEP
30 PRINT "AREA = ";WIDE*DEEP
40 END
```

Did you notice the new way to use the INPUT statement? If you put a word or string in quotation marks followed by a comma between INPUT and the variable name then the computer will print it. This is neater and saves a program line.



HINT HINT In maths, we would understand **2D** to mean **2 x D**. In BASIC we *must* have the multiply sign as **2*D** for the computer to be able to understand.

Now let's add to this program. Add this line:

```
40 PRINT "TOTAL LENGTH OF ALL SIDES =
";2*(WIDE+DEEP)
50 END
```

Try it with a few lengths.

Now we can change it to give the volume of a solid. Change these lines:


```

30 INPUT "HOW HIGH",HIGH
40 PRINT "VOLUME = ";WIDE*DEEP*HIGH
50 PRINT "SURFACE AREA = ";2*(WIDE*DE
EP+DEEP*HIGH+WIDE*HIGH)
60 END

```

To measure circles, your computer recognises the word PI for 3.14159265, so you don't have to remember exactly what it is. Here's a simple program to measure a circle:

```

10 INPUT "RADIUS",CRADIUS
20 PRINT "AREA = ";PI*CRADIUS^2
30 PRINT "CIRCUMFERENCE = ";2*PI*CRADIUS
40 END

```

If you've already learnt how to measure areas and solids in school then you'll be able to think of lots more things to try.

Now try this program (press **BREAK** first):

```

10 PRINT "HOW MUCH POCKET MONEY PER W
EEK (PENCE)"
20 INPUT MONEY
30 PRINT "IN 1 YEAR YOU WILL GET £";M
ONEY*52/100
40 WEEKS=10000/MONEY
50 PRINT "IT WILL TAKE YOU ";WEEKS;"
WEEKS TO SAVE £100"
60 END

```

Sometimes the answers in programs like this have lots of decimal places which look untidy and confusing on the screen. You can control this using another BASIC statement which your computer understands, INT (which stands for INTEGER). What's an **integer**? It just means a whole number, that is, one without any fractions or decimal places.

Try this:

```
P. INT(2.5843)
```

Note that the number after INT must be in brackets. The answer will be 2.

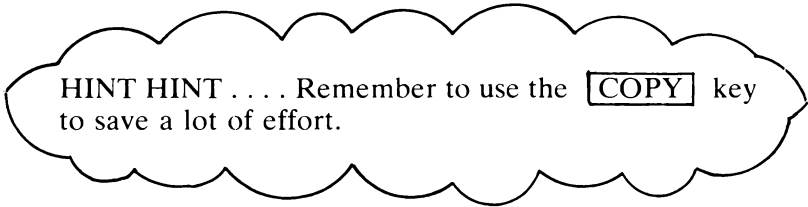
Using INT you will get the nearest whole number below the one that you put in. Now change line 40 of the Pocket Money program to:

```
40 WEEKS=INT(10000/MONEY)
```

Of course, the answer may not be strictly accurate, because if the original answer was (say) 199.9999 weeks, then the new answer will be 199 weeks. So if you were planning to have saved a certain amount by a certain date, then you could still be short of cash when that day arrived. So it may be safer to make line 40 read:

```
40 WEEKS=INT(10000/MONEY)+1
```

Here's a program that will print out a multiplication table.



HINT HINT . . . Remember to use the **COPY** key to save a lot of effort.

```
10 INPUT "NUMBER ";NUMBER
20 PRINT NUMBER;" TIMES TABLE"
30 PRINT
40 PRINT "1 X ";NUMBER;" = ";NUMBER
50 PRINT "2 X ";NUMBER;" = ";NUMBER*2
60 PRINT "3 X ";NUMBER;" = ";NUMBER*3
70 PRINT "4 X ";NUMBER;" = ";NUMBER*4
80 PRINT "5 X ";NUMBER;" = ";NUMBER*5
90 PRINT "6 X ";NUMBER;" = ";NUMBER*6
100 PRINT "7 X ";NUMBER;" = ";NUMBER*7
110 PRINT "8 X ";NUMBER;" = ";NUMBER*8
120 PRINT "9 X ";NUMBER;" = ";NUMBER*9
130 PRINT "10 X ";NUMBER;" = ";NUMBER*10
140 END
```

RUN this. Note that this program will not only work with integers (1, 2, 3, etc) but also with numbers with decimal places (1.3, 2.75, 3.142, etc). Try a few of these.

In a computer program, the method that we use to solve a problem is known as the **algorithm**.

4 Branch Line

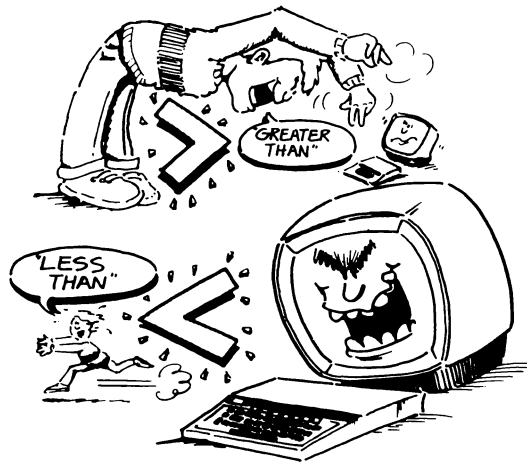
It's said that a computer can only: store and retrieve information, compare two pieces of information, and do arithmetic. We've already seen how it can store information and do arithmetic. Now we're going to use it to compare pieces of information. First of all, we need to get to know some new symbols.

= we have met already, meaning "is given the value of"
It is also used sometimes to mean "is equal to"

> means "is greater than"

< means "is less than"

<> used together mean "is not equal to"

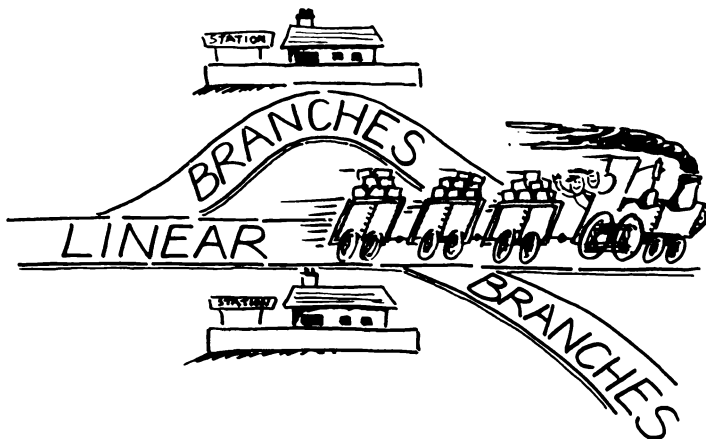


Try this:

```
10 INPUT "First number ",FIRST
20 INPUT "Second number ",SECOND
30 IF FIRST=SECOND THEN PRINT "Both nu
mbers are equal":END
40 IF FIRST>SECOND THEN PRINT FIRST;"
is greater than ";SECOND:END
50 PRINT FIRST;" is less than ";SECOND
60 END
```

HINT HINT . . . (1) If you are entering a line and it is too big to fit across the screen, don't worry, just keep on typing. Don't press **RETURN** until before the next line number. The computer will accept lines of up to 255 characters long.

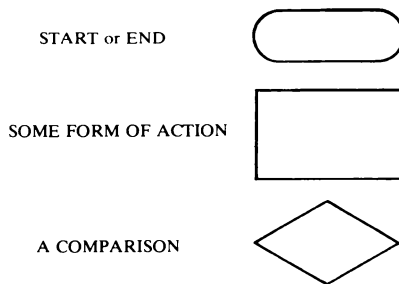
HINT HINT . . . (2) To get small letters to print on the screen (as in "First number") press the CAPS LOCK key (second up left-hand side) until the CAPS LOCK LIGHT (middle light) goes out. Don't forget to press it again (so that the light comes ON) before typing a BASIC command.



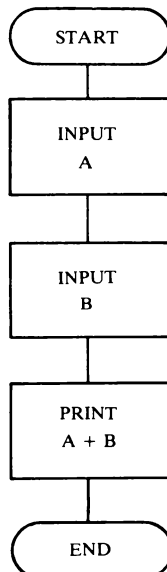
Quite a few new ideas here. First of all, all the programs that we have used up until now have been **linear**, that is the instructions were in a simple line, like a railway track. Now we have a program that **branches**. The computer makes comparisons so^s that it will know which route to follow.

Now that the program isn't so simple it will help us if we make a map – this is called a **flowchart**.

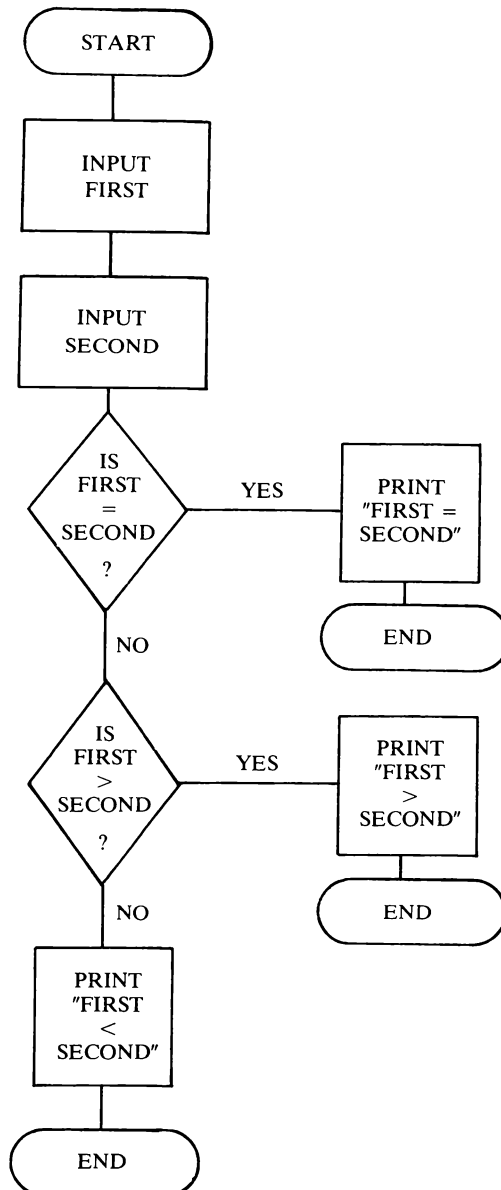
There are three types of shape on a flowchart.



Here's the flowchart of a simple program:



And here is the flowchart of the program that we have just entered into the computer:



You can see from this that the program can END in any of three places, depending on the values of A and B.

Lines 10 and 20 invite you to enter two numbers. Line 30 makes the first comparison.

```
30 IF FIRST=SECOND THEN PRINT "Both nu-
mbers are equal":END
```

This is called an IF . . . THEN statement. IF the condition described (FIRST = SECOND) is true, THEN the rest of the line is performed. If not, the program passes straight on to the next line. Note that here we have a multi-statement line: if the message is printed, then the program is ended, otherwise it would pass on to line 40.

```
40 IF FIRST>SECOND THEN PRINT FIRST;"
is greater than ";SECOND:END
```

Again, **if** the condition is true **then** the message is printed and the program ended.

If FIRST is not equal to SECOND, nor greater than SECOND, then it must be *less* than SECOND, so line 50 doesn't need to make a comparison, but just prints the message.

```
50 PRINT FIRST;" is less than ";SECOND
```

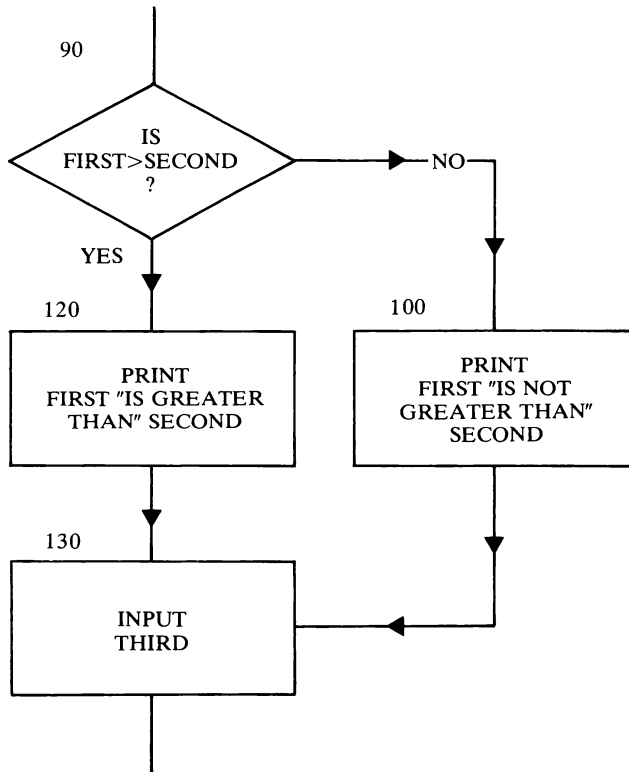
RUN this program a few times with different numbers and check that it works. Try entering some negative numbers like this:

```
-345 -346
```

You may be surprised to see that -345 is greater than -346, but -346 is the *lower* number of the two.

Another way that an IF . . . THEN statement can be used is to send the program to a different line, like this:

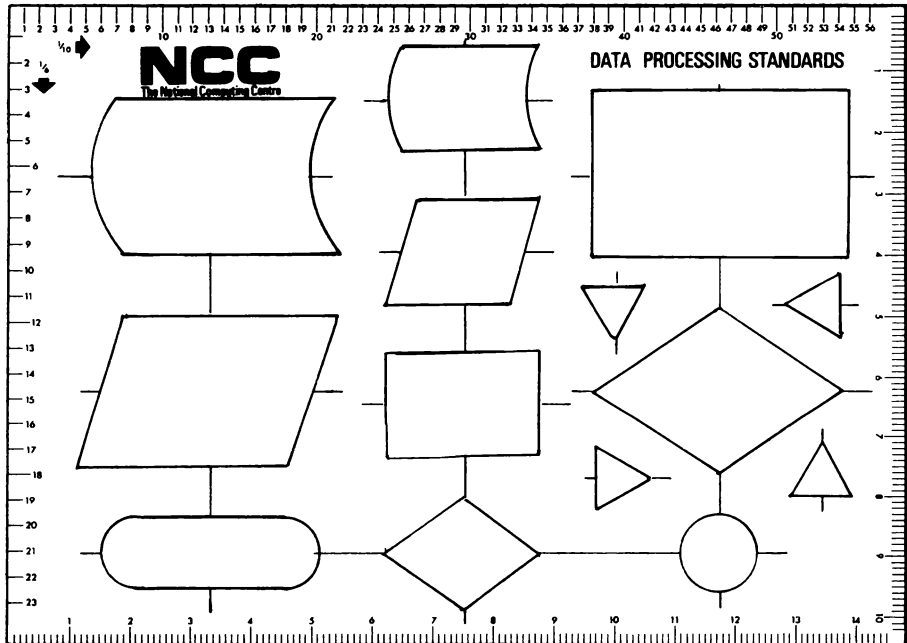
```
90 IF FIRST > SECOND THEN 120
100 PRINT FIRST;" is not greater than " ;SECOND
110 GOTO 130
120 PRINT FIRST," is greater than " ;SECOND
130 INPUT THIRD
```

In line 90, IF the condition is true, THEN the program jumps to line 120. In line 110, the GOTO statement makes the program jump to line 130 without any conditions.

When you are trying to solve a tricky problem try drawing a flowchart. Probably you'll have to cross out the first few attempts and start again, but they are really helpful in sorting out algorithms. Just a piece of paper and a pencil (and rubber) will do, but if you like things to look neat, then a flowchart template (stencil) like the one shown on page 43 will be useful.

GOTO can be a useful statement but if the jump is for more than one or two lines, or if there are more than one or two GOTOS in a program, then it can get very confusing. BBC BASIC has some more powerful commands for moving the program around which we will learn to use later.



As well as working with numbers, =, < and > also work with letters, words and strings. These are checked for equality, or alphabetical order. B would be *greater* than A, but *less* than C. Words are checked for order, as they would be in a dictionary. Do remember though, that the computer will compare words and strings very thoroughly. If the slightest detail is wrong, an extra space, or small letters instead of capitals, then the strings cannot be equal. Also, the computer cannot know that a word is spelt incorrectly, and will compare it exactly as it is.

Try this:

```

10 INPUT "First word",FIRST$
20 INPUT "Second word",SECOND$
30 IF FIRST$ = SECOND$ THEN PRINT "they are
   both the same word":END

```

```
40 IF FIRST$ < SECOND$ THEN PRINT FIRST$;"
   ";SECOND$ ELSE PRINT SECOND$;
   " ";FIRST$
50 END
```

The special interest here is line 40 which is a more powerful version of the IF ... THEN statement – an IF ... THEN ... ELSE statement.

It works like this. If FIRST\$ is less than SECOND\$, then it prints FIRST\$ before SECOND\$ so that they are in alphabetical order. If FIRST\$ is not less than SECOND\$ (and we know that it's not *equal* to SECOND, because we tested that in line 30) then SECOND is printed before FIRST.

Try the programs with different letters, words and even numbers.

Try these strings:

BOY	BOY
GREEN GRO CER	GREENGROCER
345	346
-345	-346

Two interesting things to note, GREEN GRO CER comes before GREENGROCER because a space has a lower value than a G. Also, -345 is considered *less* than -346. This is because this program is only looking at the value of the **string** and doesn't recognise the minus sign for what it is.

WORDSQUARE 1

Here is a wordsquare which contains 28 of the words which we have met so far. To make it very easy, there is a list of the words and two are shown on the square.

S	A	R	E	T	U	R	N	D	S	Y	S	V
Q	E	L	I	S	T	P	R	H	L	E	T	A
U	M	I	O	L	A	A	N	P	C	O	R	R
A	I	I	N	R	O	N	I	E	M	I	I	I
R	H	N	T	B	T	T	A	E	L	D	N	A
E	V	T	Y	U	L	N	I	B	U	G	G	B
R	C	E	H	U	P	G	O	B	R	I	S	L
O	K	G	M	T	O	N	P	C	O	P	Y	E
O	H	E	E	B	A	S	I	C	R	E	L	L
T	I	R	C	A	P	S	L	O	C	K	O	T
L	R	I	P	L	U	S	C	M	I	N	U	S
H	A	M	A	R	G	O	R	P	A	R	G	D
S	C	U	O	D	I	U	C	U	R	S	O	R
T	T	D	Q	N	N	N	A	T	N	I	T	E
L	F	R	D	E	L	E	T	E	E	E	O	D
D	I	V	I	D	E	W	S	R	M	A	N	S

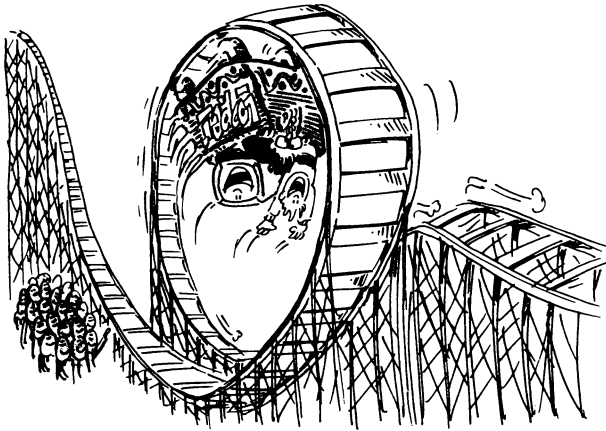
BASIC
BBCCOMPUTER
BUG
CAPSLOCK
CLS

CONTROL
COPY
CURSOR
DELETE
DIVIDE

END
EQUAL
GOTO
INPUT
INTEGER
KEYBOARD
LIST
MINUS
MULTIPLY
NEW

PLUS
PRINT
PROGRAM
RETURN
RUN
STRING
SQUAREROOT
VARIABLE

5 Looping the Loop



Now we will try a very useful function, the FOR ... TO ... NEXT loop. You know by now that your computer can do sums very quickly, far more quickly than we can type them in. So we will let it feed in the numbers itself. We'll just tell it when to start, and when to stop!

Type this in (remember to **BREAK** first):

```
10 FOR COUNTER=1 TO 10
20 PRINT COUNTER
30 NEXT COUNTER
40 PRINT "END"
50 END
```

and RUN. Your computer will print all the numbers from 1 to 10. How does this work? When it comes to line 10 in the program, the

computer will give the variable COUNTER the first value of the FOR statement. Thus, COUNTER has the value of 1, which is printed in line 20, followed by a space. When it comes to line 30, NEXT tells the computer to go back to line 10, and add 1 to the value of COUNTER. Then it checks that COUNTER is now no larger than the number following the "TO" in line 10. It keeps on going around this loop adding 1 to the value of COUNTER each time, until COUNTER is bigger than the number following the "TO". When COUNTER is bigger than the "TO" value, then the program jumps to the line following NEXT COUNTER.

Change line 10 to read:

```
10 FOR COUNTER=1 TO 100
```

RUN it to see what happens. Now try altering line 10 to read:

```
10 FOR COUNTER=1 TO 100 STEP 4
```

Do you see what happens now? Instead of adding 1 to COUNTER each time around the loop, the computer adds 4. Notice that the last number to be printed is 97 and not 100. This is because the next number after 97 would be 101, but this is bigger than 100 so the program jumps to line 40 without printing it.

HINT HINT . . . (1) When you want to LIST a program with loops, first type LISTO7 **RETURN** , and then LIST. This way the computer will add extra spaces in the listing to make it easier to read. But be careful if you use the COPY key after using LISTs. See "**Wrong spaces**" in Appendix 1.

HINT HINT . . . (2) Instead of typing out NEXT each time, just type N.

Now try these:

```
10 FOR COUNTER=1 TO 100 STEP 2.75
10 FOR COUNTER=51.7 TO 58.9 STEP 0.42
10 FOR COUNTER=20 TO 4 STEP -1
```

Now let's use a FOR. . .NEXT loop to do something useful. We'll use it to improve the Times Table program which we wrote in Chapter 3. Try this:

```
10 INPUT "WHICH TIMES TABLE",TIMES
20 CLS
30 PRINT TIMES;" TIMES TABLE"
40 PRINT
50 FOR COUNTER=0 TO 10
60 PRINTTIMES;" X ";COUNTER;" = ";
TIMES*COUNTER
70 NEXT COUNTER
80 END
```

This is much shorter than the previous program. Try it with a few different values, then try changing the **loop parameter** in line 50. For example try these:

```
50 FOR COUNTER=247 TO 264
50 FOR COUNTER=10 TO 0 STEP-1
```

Did you also see CLS in line 20? This clears the screen. It has the same effect as CONTROL-L, but can be used in a program.

Now we'll try an extra feature of FOR. . .NEXT loops, that is NESTED LOOPS. These are loops inside loops, like nested boxes. First, alter the last program so that it is back to how it was originally (50 FOR COUNTER = 0 TO 10). Then change lines 10 and 80 and add lines 90 to 130 so that it becomes:

```
10 FOR TIMES=0 TO 10
20 CLS
30 PRINT TIMES;" TIMES TABLE"
40 PRINT
50 FOR COUNTER=0 TO 10
```



```

    60 PRINTTIMES;" X ";COUNTER;" = ";TIM
    ES#COUNTER
    70 NEXT COUNTER
    80 PRINT
    90 IF TIMES=10 THEN 120
    100 PRINT "FOR NEXT TABLE PRESS ANY KE
    Y"
    110 X$=GET$
    120 NEXT TIMES
    130 END

```

Now we have two loops, the one that we had last time from lines 50 to 70 which works out the lines for one table, and the larger loop from lines 10 to 130 (the whole program in fact) which gives the number for each table.

There are many uses for nested loops, and you will come across them when you type in programs from magazines, etc.



The important thing to remember is that every FOR must have a NEXT, and every NEXT must have a FOR. If not, the program won't work and you may get an error message from the computer which says:

NEXT WITHOUT FOR ERROR IN LINE . . .

which can be very helpful for finding where you have gone wrong.

Remember, in order to see your loops more easily when you LIST the program, type LISTO7 **RETURN** LIST **RETURN** . This will list the program on the screen with each level of nested loops set in (indented) so that they can be more easily checked.

FOR. . .NEXT loops are very common, especially in programs printed in magazines. This is because FOR. . .NEXT loops can be used on all personal computers. But the BBC and Electron computers understand another similar command which is very useful. This is REPEAT . . . UNTIL. This works like the FOR . . . NEXT loop except that we don't need to know in advance how many times the program must go round the loop. For example:

```
10 REPEAT
20 randomnumber=RND(10)
30 PRINT ;randomnumber;" * ";
40 UNTIL randomnumber=9
```

HINT HINT . . . (1) Notice that the variable **random-number** has been entered in lower case (small letters). This is useful because it is then much easier to tell the difference between **variable names** and **BASIC words** (which *have* to be in capital letters).

HINT HINT . . . (2) RND is explained at the end of the chapter.

REPEAT. . .UNTIL loops are always performed at least once. They are a bit faster than FOR. . .NEXT loops, although you will only notice this if the program goes around the loop many times.

As with FOR. . .NEXT, typing LISTO7 RETURN LIST RETURN will show up the REPEAT. . .UNTIL loops. REPEAT. . .UNTIL loops can be nested like FOR. . .NEXT loops.

There is another new trick which has been used in the Times Tables program. Did you notice it? It's in lines 100 and 110.

If you want the computer to wait until you are ready, then these lines can be very useful.

```
110 X$ = GET$
```

GET\$ is an input from the keyboard but in this case it's not necessary to press **RETURN**, the computer waits for a key to be pressed and then accepts the character as soon as it's input.

In this case it doesn't matter what character is entered, because it is just a signal to the computer that it can carry on.

You will have seen that the computer performs its various functions very quickly (and the BBC computer is one of the fastest to work in BASIC). But there are times when it is important to get it to work even faster. One example is when processing a lot of data, to do thousands of calculations. Another is in computer games with moving graphics. Here, each movement on the screen will need the computer to do many calculations and loops. To speed things up it is possible to use **integer variables**. We have already met **integers** in Chapter 3. They are whole numbers, without fractions or decimal parts. If we tell the computer that a variable is only going to contain an **integer** number, then it can work much faster. It also uses less memory space. To do this we label the variable with %. So **A%** or **number%** are **integer variables**, whereas **A** or **number** would be ordinary ones (otherwise known as **floating point** variables, because the decimal point can be anywhere in the number).

Try this:

```
10 CLS
20 PRINT''' "Calculating"
30 TIME=0
40 J%=0
50 FOR I%=1 TO 1000
60 J%=J%+I%
70 NEXT I%
80 time1%=TIME
90 TIME=0
100 J=0
110 FOR I=1 TO 1000
120 J=J+I
130 NEXT I
140 time2=TIME
150 PRINT''' time1%/100,time2/100
160 PRINT"Integer variables work in "
170 PRINT;INT(time1%/time2*100);
```

This program has two loops, and goes around each 1000 times. During the first loop I% is added to J% (in line 60). In the second loop, I is added to J (line 120).

TIME is the computer's internal clock. In lines 30 and 90 it is zeroed then it will count up in 1/100ths of a second until zeroed again. This is an example where our variables **time1%** and **time2** need to be in **lower case letters** to avoid confusion with a BASIC word.

Time1% and time2 will be shown in seconds and you will probably find that the integer loop runs about twice as fast as the other. Try altering the loops to go round 10,000 times and see if there is any difference.

By the way, did you notice the four apostrophies (') in line 20? Each causes the cursor to jump down a line so that the text is lower down the screen.

RND

RND was shown in the short program on page 51. This is a BASIC statement that you will come across many times, especially in games programs. RND is short for RANDOM and will give a random number.

RND(1) will give a number anywhere between 0 and 0.999999.
RND(10) will give a integer number anywhere between 1 and 10.
RND(100) will give an integer number anywhere between 1 and 100.

Any other number in the brackets will give an integer number anywhere between itself and 1.

Try this:

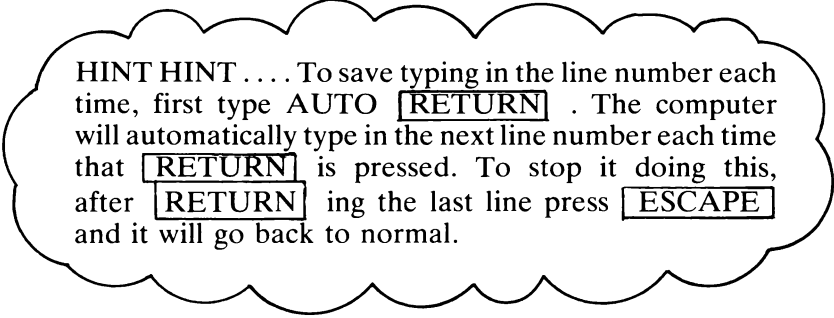
```
10 INPUT "INPUT NUMBER",NUMBER
20 FOR I=1 TO 100
30 PRINT ;RND(NUMBER);" / ";
40 NEXT
50 PRINT
60 END
```

RND is very useful where you want a "surprise" number and is used in games and any program where you need "artificial" data.

6 Hip Hip Array

Computer programs deal with **information**. This needs to be stored in the computer's memory and then manipulated during the program RUN. In this chapter we are going to look at ways of arranging this information in **arrays** so that it will be easy to rearrange, and then we will look at a way of getting it into the memory when we load the program using DATA statements.

First of all let's create some values. Type in this program and RUN it, we'll look at how it works afterwards.



HINT HINT To save typing in the line number each time, first type AUTO **RETURN** . The computer will automatically type in the next line number each time that **RETURN** is pressed. To stop it doing this, after **RETURN** ing the last line press **ESCAPE** and it will go back to normal.

Try this:

```
10 DIM A(4)
20 FOR I=1 TO 4
30 A(I)=RND(10)
40 NEXT
50 FOR I=1 TO 4
60 PRINTA(I)
70 NEXT
80 END
```

Type RUN and the result will be something like this (but different every time).

```
1      5      2      7
```

Well, a bit of explanation is needed here. What we have done is to set up an **array**. That is, a list of variables, all with the same name, but each followed by a number in brackets A(1), A(2), A(3), A(4).

Computer people call these by a long name – **subscripted variables**, which just means that mathematicians would have written the numbers *below* the line like this A_1 , A_2 , A_3 , etc. It's not convenient to do this on a computer so we put the numbers in brackets, but still use the long name.

Now let's look at each line in turn.

```
10 DIM A(4)
```

When we are going to use an array, we must warn the computer in advance, so that it can leave enough storage space in the memory. We use a DIM (dimension) statement. If we want to use an array of 20 variables we would use DIM A(20). If you need to save space you *could* use DIM A(19). Because a computer starts counting at 0, 0–19 gives 20. If we haven't dimensioned enough space then when we RUN the program we will get the error message: ARRAY. You must put the DIM statement right at the beginning of the program because you cannot change it during a RUN.

```
20 FOR I = 1 TO 4
```

This is a simple loop parameter as we used in Chapter 5.

```
30 A(I) = RND(10)
```

Each time the program goes round the loop the value of I will increase by 1 until it gets to 4. So in this line we **declare** the variables in the array, as the program goes around the loop the values will be:

```
A(1) = RND(10)
```

```
A(2) = RND(10)
```

```
A(3) = RND(10)
```

```
A(4) = RND(10)
```

Lines 50 to 70 are another loop which prints out the values in the array.

Now alter the program and add more lines so that it is like this (using the COPY key will save a lot of typing):

```
10 DIM A(4),B(4)
20 FOR I=1 TO 4
30 A(I)=RND(10):B(I)=RND(10)
40 NEXT
50 FOR I=1 TO 4
60 PRINTA(I);
70 NEXT
80 PRINT
90 FOR I=1 TO 4
100 PRINTB(I);
110 NEXT
120 PRINT
130 FOR I=1 TO 4
140 PRINTA(I)+B(I);
150 NEXT
160 END
```

Now RUN the program and the result will look something like this:

>RUN

4	3	6	1
7	7	7	2
11	10	13	3

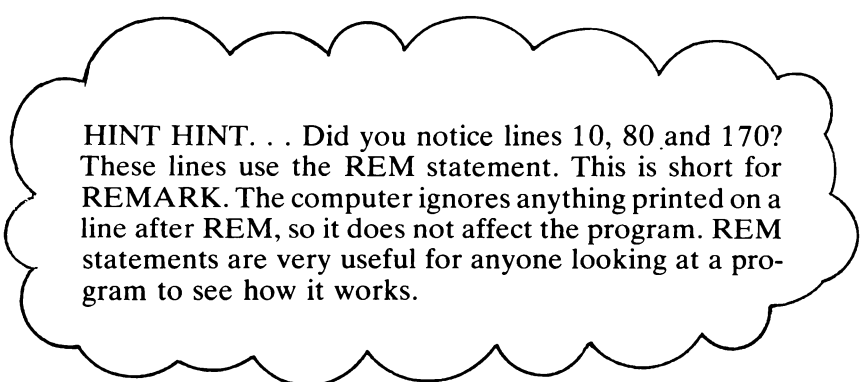
Now we have *two* subscripted variables, A and B. Each array contains four random numbers. Lines 50 to 70 print out A. Lines 80 to 100 print B. Lines 130 to 150 print out A+B.

So the subscript (the number in brackets) can be computed, just as the values of the variables can. This is a very important feature which is used in a lot of programs.

Now try this:

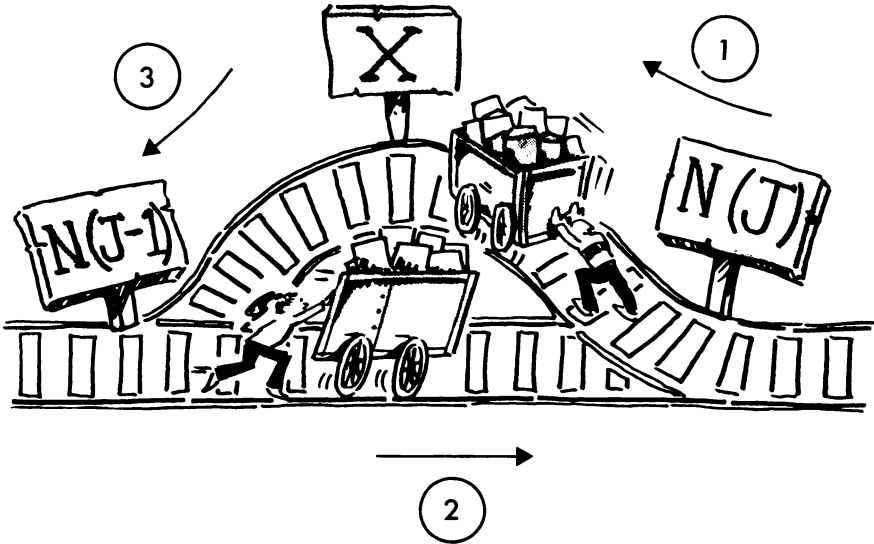
```
10 REM *** SORTING NUMBERS ***
20 DIM N(5)
30 PRINT"ENTER 5 NUMBERS"
40 FOR I=1 TO 5
50 PRINT"NUMBER ";I;
60 INPUT N(I)
70 NEXT
80 REM *** SORT NUMBERS OUT ***
90 FOR I=5 TO 2 STEP -1
100 FOR J=2 TO I
110 IF N(J)>N(J-1) THEN 150
120 X=N(J)
130 N(J)=N(J-1)
140 N(J-1)=X
150 NEXTJ
160 NEXTI
170 REM * PRINT OUT SORTED NUMBERS *
180 FOR I=1 TO 5
190 PRINTN(I)
200 NEXTI
210 END
```

This is a sort routine which works by swapping around the contents of an array $N(1) \dots N(5)$. The program itself is from lines 90 to 160. There are two nested loops and these compare array items next to each other and if a larger item comes before a smaller one, it swaps them around.



HINT HINT. . . Did you notice lines 10, 80 and 170? These lines use the REM statement. This is short for REMARK. The computer ignores anything printed on a line after REM, so it does not affect the program. REM statements are very useful for anyone looking at a program to see how it works.

As a computer can only do one thing at a time, it's not possible to swap two items directly, so in lines 120 and 140, the variable X is used like a railway siding to shunt one number into while the other is swapped.



This type of sort routine is called a **bubble sort**. There are many other different ways of sorting, but this is a good general-purpose method. It can be adapted for larger lists, or for dealing with strings.

For example, if we want it to handle a different number of numbers then we would have to change these lines (remember to use the COPY and cursor control keys to save typing).

```

15 INPUT "HOW MANY NUMBERS", howmany
20 DIM N(howmany)
30 PRINT "ENTER "; howmany; " NUMBERS"
40 FOR I=1 TO howmany
50   FOR J=howmany TO I STEP -1
60     IF N(J) < N(J-1) THEN
70       X=N(J)
80       N(J)=N(J-1)
90       N(J-1)=X
100    END IF
110  END FOR
120 END FOR

```

And to handle letters, words or strings to put them into alphabetical order (as you would find in a dictionary for instance), make these changes:

```

10 REM ** ENTER NAMES **
15 INPUT "HOW MANY NAMES", howmany
20 DIM N$(howmany)
30 PRINT "ENTER "; howmany; " NAMES"
40 FOR I=1 TO howmany
50 PRINT "NAME "; I;
60 INPUT N$(I)
70 NEXT
80 REM ** SORT NAMES **
90 FOR I=howmany TO 2 STEP -1
100 FOR J=2 TO I
110 IF N$(J)>N$(J-1) THEN 150
120 X$=N$(J)
130 N$(J)=N$(J-1)
140 N$(J-1)=X$
150 NEXT J
160 NEXT I
170 REM ** PRINT OUT SORTED NAMES **
180 FOR I=1 TO howmany
190 PRINT N$(I)
200 NEXT I
210 END

```

In the next chapter there's a program to sort out different items into different orders.

Many programs, and especially many games programs, need to have work numbers that can be used during the run. If we have to store these as variables, like this:

```
20 A=27.3 : B=4 : C=321 : D=4000 : A#=600M
```

then it would be very inconvenient, we would use up a lot of memory space. We can store this sort of information using DATA statements. These are written into the program and although we can change the programs whenever we want to, the DATA state-

ments cannot be altered by the computer program itself. Therefore, every time that you RUN the program you will get the same DATA. The DATA lines can be anywhere in the program, the computer looks for them when needed.

Try this:

```
10 DIM day$(7)
20 FOR I=1 TO 7
30 READ day$(I)
40 NEXT I
50 PRINT"The days of the week are"
60 FOR I=1 TO 7
70 PRINT"DAY ";I;" = ";day$(I)
80 NEXT I
90 DATA Sunday, Monday, Tuesday, Wednesday,
Thursday, Friday, Saturday
```

RUN the program.

In line 30 the READ statement is rather like INPUT except that the computer looks for the item in DATA lines, starting with the first item it finds (if it's *not told* to start with a different one) and then taking them one by one in order. Each data item is separated by a comma (,), but there is no comma at the end of a DATA line (or else the computer will think that there is an extra item – a blank space – at the end of the line. You can try adding a comma after Sunday in line 15 below to test this).

Now change these lines:

```
15 DATA Sunday
25 DATAMonday, Tuesday
35 DATAWednesday
45 DATAThursday, Friday, Saturday
90 RETURN
```

LIST the program to check that it has altered and then RUN it. You'll see that there is no difference. The computer keeps a check on the DATA that has been read and looks for the next item. This

works even if there are several READ statements in different parts of the program.

You will need to be careful not to run out of DATA or the program will crash. See what happens if you try to print out a fortnight. Change these lines:

```
10 DIM day$(14)
20 FOR I=1 TO 14
60 FOR I=1 TO 14.
```

When the program runs out of DATA to read you will get the error message "out of data". Now add these lines:

```
31 IF day$(I)="endofdata" THEN RESTORE:I=I-1
46 DATAendofdata
```

Now we have added an extra DATA item which the program can detect. When the program comes to this item it RESTOREs the DATA pointer back to the first DATA item. It is reduced by 1 so that "endofdata" is replaced in the array by the next proper DATA item.

DATA lines can contain either numbers or strings, or even a mixture of the two. They must be READ correctly.

Try this:

```
10 DIM word$(4),number(4)
20 FOR I=1 TO 4
30 READ word$(I),number(I)
40 PRINT I,word$(I),number(I)
50 NEXT
60 DATAapple,23,boy,10
70 DATAcat,73,do9,58
```

RUN this and check that it works. Then change line 30 to read:

```
30 READ number(I),word$(I)
```

7 How To Proceed

In this chapter we are going to build up a larger program than we have up to now. We will enter it a section at a time and check that each is working correctly as we go along.

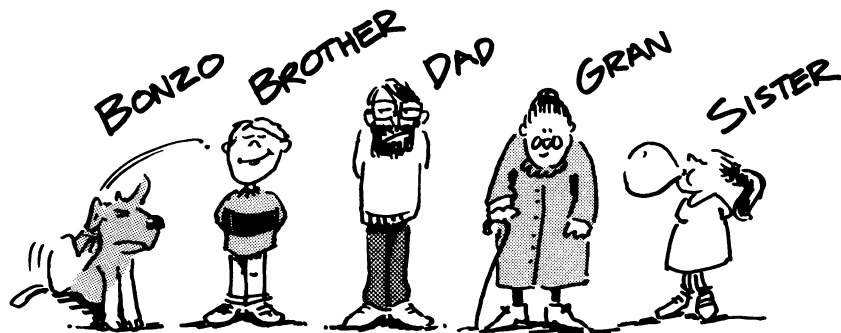
To do this we will build the program from blocks, which in BBC BASIC are called PROCEDURES.

Note that some program statements are so long that they spill over onto the next line. *Don't* press **RETURN** until you come to the next line number.

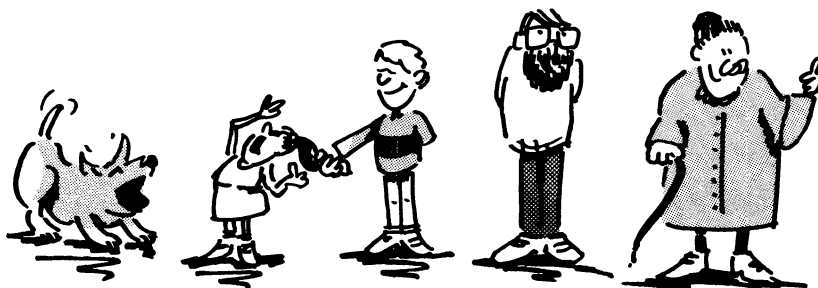
The program that we are going to enter will accept a list of names, ages and heights and sort these lists into alphabetical, age and height order.

First of all, key this in:

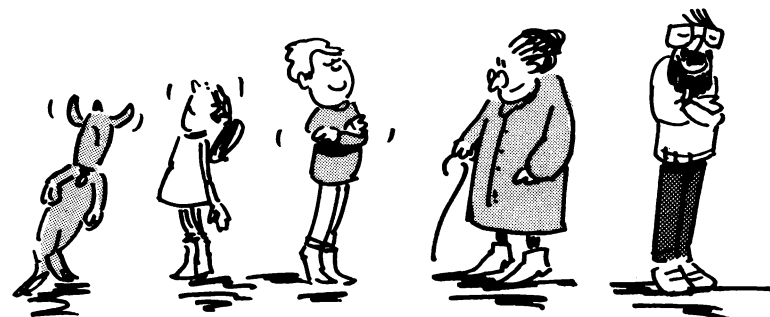
```
10 REM **PROGRAM TO SORT LISTS**
20 DIM name$(20),age(20),height(20)
30 number=0
290 PRINT'"Enter name (family name onl
y)"
```



Alphabetical Order



Order of Age



Order of Height

```

300 number=number+1
310 IF number>20 THEN PRINT "LIST FULL"
:GOTO 370
320 PRINT "NAME ";number;" (enter XX to
quit)"
330 INPUT name$(number): IF name$(number)
)="XX" OR name$(number)="xx" THEN 370
340 INPUT "Age(years) ",age(number)
350 INPUT "Height(cms) ",height(number)
360 GOTO 300
370 number=number-1
810 PRINT "NAME";TAB(25)"AGE";TAB(30)"H
EIGHT"
820 FOR I=1 TO number
830 PRINTTAB(5)name$(I);TAB(25)age(I);T
AB(30)height(I)
840 NEXT I

```

HINT HINT . . . Did you notice that in line 310 there are two statements on the same line? You can do this here if they are separated by a colon (:). This has been done because we only want to GOTO 370 *if* number >20. This is called a **multi-statement** line. You can also do it to save space in long programs.

So far the program only lets you enter the names, ages and heights and then prints out the list.

Now we will turn these two routines into PROCEDURES. Type in lines 60, 100, 110, 280, 380, 790, 850 and 860 so that the program is as shown below:

```

10 REM **PROGRAM TO SORT LISTS**
20 DIM name$(20),age(20),height(20)
30 number=0
60 PROCcenter
100 PROCprint

```



```

110 END
280 DEF PROCcenter
290 PRINT'"Enter name (family name only)"
300 number=number+1
310 IF number>20 THEN PRINT "LIST FULL"
:GOTO 370
320 PRINT'"NAME ";number;" (enter XX to quit)"
330 INPUT name$(number):IF name$(number)
="XX" OR name$(number)="xx" THEN 370
340 INPUT"Age(years) ",age(number)
350 INPUT"Height(cms) ",height(number)
360 GOTO300
370 number=number-1
380 ENDPROC
790 DEF PROCprint
810 PRINT'"NAME";TAB(25)"AGE";TAB(30)"H
EIGHT"'
820 FORI=1 TO number
830 PRINTTAB(5)name$(I);TAB(25)age(I);T
AB(30)height(I)
840 NEXTI
850 PRINT
860 ENDPROC

```

The line numbers may seem to be all over the place but they will begin to make sense later.

Now we can begin to see how PROCEDURES are used. The main program is now in lines 60 and 100. In line 60, the PROCEDURE called PROCcenter is called. On coming to this line, the computer searches through the program for a PROCEDURE of this name. PROCcenter is defined in lines 280 to 380. It starts with the statement DEF PROCcenter and ends with ENDPROC. Similarly in line 100, PROCprint is called, and this is defined in lines 790 to 860.

When the computer reaches an ENDPROC statement it returns to the program line where it was when the PROCEDURE was called. Note that END must be used at the end of the main

HINT HINT . . . We now have a program that is too long to fit on the screen all at once. The computer LISTs so fast that there is no hope of reading it as it flashes past. Try typing LIST, 380 **RETURN** . All the lines up to 380 will be LISTed. Now try LIST 380, **RETURN** and all the lines from 380 on will appear. You can also try LIST 100, 400 **RETURN** . This is fine if you know where you want to look. With a long program it could still be tedious looking for the lines that you want. Press **CONTROL** and N together, and then type LIST **RETURN** . The computer will now list the first screenful of the program. To see the next screenful, press **SHIFT** , and so on. It's normal for both the SHIFTLOCK and CAPS LOCK lights to be lit when doing this.

program (unless it goes around a loop), otherwise the computer will carry on through the program and try to perform the first procedure again.

Now we'll add the first sort routine. Add the lines 400 to 490 and 730 to 770 so that the program is as follows:

```

10 REM **PROGRAM TO SORT LISTS**
20 DIM name$(20), age(20), height(20)
30 number=0
40 PROCcenter
50 PROCnames
60 END
70 DEF PROCcenter
80 PRINT""Enter name (family name o
nly)"
90 number=number+1
100 IF number>20 THEN PRINT "LIST FUL
L":GOTO 370

```

```

320 PRINT "NAME ";number;" (enter XX
to quit)"
330 INPUT name$(number):IF name$(number)="XX" OR name$(number)="xx" THEN 370
340 INPUT "Age(years) ",age(number)
350 INPUT "Height(cms) ",height(number)
)
360 GOTO300
370 number=number-1
380 ENDPROC
400 DEF PROCnames
410 FOR I=number TO 2 STEP-1
420 FOR J=2 TO I
430 IF name$(J)>name$(J-1) THEN 450
440 PROCswaP
450 NEXT J
460 NEXT I
470 legend$="alphabetical order"
480 PROCprint
490 ENDPROC
730 DEF PROCswaP
740 X$=name$(J):XA=age(J):XH=height(J)
)
750 name$(J)=name$(J-1):age(J)=age(J-1):height(J)=height(J-1)
760 name$(J-1)=X$:age(J-1)=XA:height(J-1)=XH
770 ENDPROC
790 DEF PROCprint
800 PRINT"" "List Printed in ";legend$
$
810 PRINT "NAME";TAB(25)"AGE";TAB(30)
"HEIGHT"
820 FORI=1 TO number
830 PRINTTAB(5)name$(I);TAB(25)age(I)
TAB(30)height(I)
840 NEXTI
850 PRINT
860 ENDPROC

```

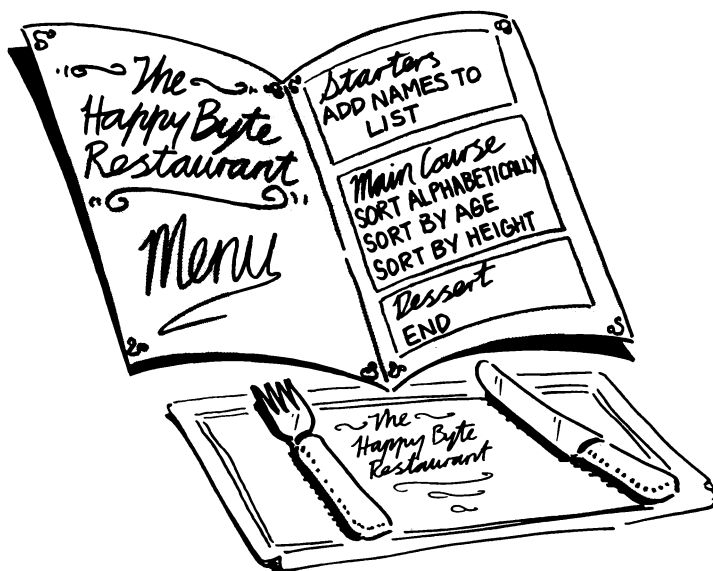
PROCprint is now called within PROCnames. So is PROCswap. So procedures can be nested like FOR. . .NEXT and REPEAT . . .UNTIL loops. The main advantage of this is the saving of program lines. The same procedure can be called from several other procedures and so only has to be keyed in once.

PROCswap may seem rather complicated at first glance, but in fact is no different from the simple sort routine in the last chapter. Here, all three lists (name\$, age and height) have to be swapped at the same time, or else things would get very chaotic.

Now add these extra lines:

```
80 PROCages
90 PROCheights
510 DEF PROCages
520 FOR I=number TO 2 STEP -1
530 FOR J=2 TO I
540 IF age(J)>age(J-1) THEN 560
550 PROCswap
560 NEXTJ
570 NEXTI
580 legend$="order of age"
590 PROCprint
600 ENDPROC
620 DEF PROCheights
630 FOR I=number TO 2 STEP -1
640 FOR J=2 TO I
650 IF height(J)>height(J-1) THEN 670
660 PROCswap
670 NEXTJ
680 NEXTI
690 legend$="order of height"
700 PROCprint
710 ENDPROC
```

Now we can sort the list in three different ways, all using the same PROCswap.



Now, to finish the program off, we can add a menu (as in a restaurant) so that the user can choose what the program will do next. Add lines 40 to 270 so that the program is as follows: (This is a complete listing so that you can check everything.)

```

10 REM **PROGRAM TO SORT LISTS**
20 DIM name$(20),age(20),height(20)
30 number=0
40 REPEAT
50 PROCmenu
60 IF select=1 THEN PROCenter
70 IF select=2 THEN PROCnames
80 IF select=3 THEN PROCages
90 IF select=4 THEN PROCheights
100 IF select=5 THEN PRINT""End of
Program." :END
110 PRINT"Press any key to return to
the menu."
120 continue$=GET$
130 UNTIL FALSE
140 REM*****

```

```
150 DEF PROCmenu
160 CLS
170 PRINTTAB(5)''' "Select required fu
nction"
180 PRINT' "(1) ADD NAMES TO LIST"
190 PRINT' "(2) SORT ALPHABETICALLY"
200 PRINT' "(3) SORT BY AGE"
210 PRINT' "(4) SORT BY HEIGHT"
220 PRINT' "(5) END PROGRAM"
230 PRINT' "Enter number required"
240 select=VAL(GET$)
250 IF select<1 OR select>5 THEN 230
260 ENDPROC
270 REM*****
280 DEF PROCcenter
290 PRINT' "Enter name (family name o
nly)"
300 number=number+1
310 IF number>20 THEN PRINT "LIST FUL
L":GOTO 370
320 PRINT' "NAME ";number;" (enter XX
to quit)"
330 INPUT name$(number): IF name$(numb
er)="XX" OR name$(number)="xx" THEN 370
340 INPUT "Age(years) ",age(number)
350 INPUT "Height(cms) ",height(number
)
360 GOTO300
370 number=number-1
380 ENDPROC
390 REM*****
400 DEF PROCnames
410 FOR I=number TO 2 STEP-1
420 FOR J=2 TO I
430 IF name$(J)>name$(J-1) THEN 450
440 PROCswap
450 NEXTJ
460 NEXTI
470 legend$="alphabetical order"
480 PROCprint
```

```

490 ENDPROC
500 REM*****
510 DEF PROCages
520 FOR I=number TO 2 STEP -1
530 FOR J=2 TO I
540 IF age(J)>age(J-1) THEN 560
550 PROCswap
560 NEXTJ
570 NEXTI
580 legend$="order of age"
590 PROCprint
600 ENDPROC
610 REM*****
620 DEF PROCheights
630 FOR I=number TO 2 STEP -1
640 FOR J=2 TO I
650 IF height(J)>height(J-1) THEN 670
660 PROCswap
670 NEXTJ
680 NEXTI
690 legend$="order of height"
700 PROCprint
710 ENDPROC
720 REM*****
730 DEF PROCswap
740 X$=name$(J):XA=age(J):XH=height(J)
)
750 name$(J)=name$(J-1):age(J)=age(J-1):height(J)=height(J-1)
760 name$(J-1)=X$:age(J-1)=XA:height(J-1)=XH
770 ENDPROC
780 REM*****
790 DEF PROCprint
800 PRINT""List Printed in ":legend$
$
810 PRINT"NAME";TAB(25)"AGE";TAB(30)
"HEIGHT"
820 FORI=1 TO number

```

```
830 PRINTTAB(5)name$(I);TAB(25)age(I)
;TAB(30)height(I)
840 NEXTI
850 PRINT
860 ENDPROC
```

This program can be adapted to many uses in your class, or club or team. The lists don't have to be of ages and heights, but can be of scores, or running times (you may want to sort these into reverse order). The number of people that the list can hold can be increased by simply changing the numbers in the DIM statements in line 20 and in line 310. It doesn't matter if you enter fewer names than the number allowed for in the DIM statement, but if the list is too big you may run out of memory space.

The great advantage of **PROCEDURES** is that they allow you to break up your program into parts, each of which can be worked on, improved and checked without affecting the rest of the program. Professional programmers consider this to be "a good thing" and the whole idea is known as **Structured Programming**.

The point is that although you are writing your program for the computer to understand, it's very important that *you* can understand it too. And anyone else that might need to read it as well. A long complicated program, written all in one piece, with lots of GOTOs flying off in all directions is almost impossible for a mere human to understand when it works, let alone debug if it goes wrong.

PROCEDURES can be a real help in writing good programs. They break the program up into easily-managed parts and they are easy to recognise when called because they can have meaningful names.

PROCEDURES are only likely to be found in programs written specially for the BBC computer and the Electron. Other versions of BASIC use a similar feature (which will also work on your machine) called **SUBROUTINES**. Instead of calling a **PROCEDURE** by a name that is easy to recognise, a **SUBROUTINE** has

to be called by its line number (which can get rather confusing in a long program).

Try this:

```
10 FOR I=1 TO 20
20 GOSUB 70
30 PRINT" line 30 ";
40 GOSUB 90
50 NEXT I
60 END
70 PRINT;I,"line 70 ";
80 RETURN
90 PRINT"line 90 "
100 RETURN
```

GOSUB causes the program to jump to the line shown. RETURN will return the program to the point immediately after the GOSUB. Again, don't forget END, or else the program will run on as far as line 70 and print:

RETURN WITHOUT GOSUB at line 70.

8 Keeping Tabs On Things

In this chapter we'll look at how text can be moved around the screen.

First we'll look at TAB. We already used TAB in the last chapter. It wasn't explained then but you probably picked up how it works. TAB (short for TABulate) tells the computer how many spaces from the left-hand side of the screen it has to start PRINTing.

Try this:

```
10 X$="HERE"  
20 PRINTX$  
30 PRINTTAB(10)X$  
40 PRINTTAB(20)X$;TAB(30)X$
```

In line 20, HERE is printed at the left-hand side of the screen. In line 30 it is printed ten spaces from the left. In line 40 it is printed 20 spaces from the left and then *again* 30 spaces from the left.

Now try this:

```
10 X$="HERE"  
20 FOR I=1 TO 16  
30 PRINT TAB(I)X$;TAB(37-I)X$  
40 NEXT I
```

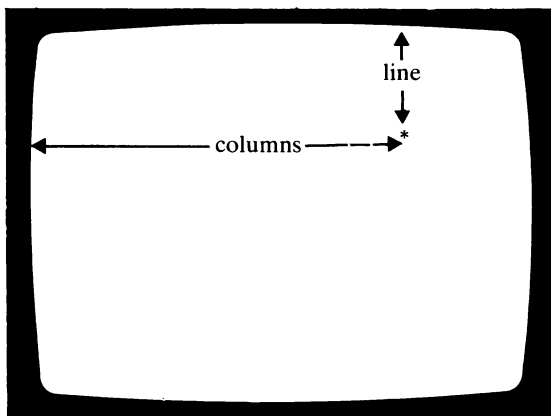
So you can see that TAB can be calculated.

So far we have only used TAB to position the printing in from the left-hand side of the screen, but it can also be used to position the print anywhere on the screen.

Try this:

```
10 CLS
20 PRINTTAB(25,10)"*"
```

Here we have used TAB (COLUMN, LINE). COLUMN is the number of columns in from the left-hand side of the screen and LINE is the number of lines down from the top.



Now try this:

```
10 CLS
20 FOR I=1 TO 40
30 PRINTTAB(RND(40),RND(24));I
40 NEXT
```

Another useful feature for positioning print is LEN.

Try this:

```
10 X$="CHRISTOPHER"
20 Y$="STEPHEN"
30 PRINTX$;" = ";LEN(X$);" LETTERS"
40 PRINTY$;" = ";LEN(Y$);" LETTERS"
```

```
>RUN
CHRISTOPHER = 11 LETTERS
STEPHEN = 7 LETTERS
```

LEN(string) counts the number of characters in the string.

Try this:

```
10 FOR I=1 TO 6
20 READ X$
30 PRINTTAB(20-LEN(X$))X$
40 NEXT I
50 DATADOORWAY,AT,WINDOW,CUP,CHAIR,
BOOK
```

RUN this, notice that all the strings now end at the same column.

```
>RUN
DOORWAY
  AT
WINDOW
  CUP
CHAIR
BOOK
```

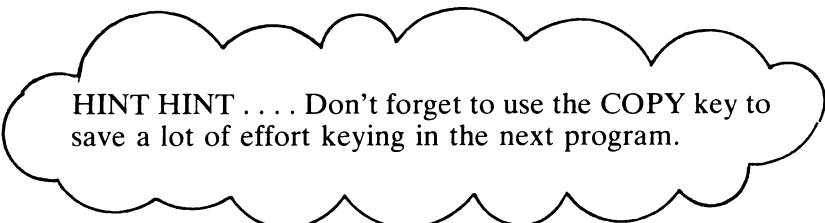
Now try altering line 30 to read:

```
30PRINTTAB(20-LEN(X$))X$" * "X$
```

```
>RUN
DOORWAY * DOORWAY
  AT * AT
WINDOW * WINDOW
  CUP * CUP
CHAIR * CHAIR
BOOK * BOOK
```

Try adding some words of your own (change the loop parameter in line 20).

Now try this:



HINT HINT . . . Don't forget to use the COPY key to save a lot of effort keying in the next program.

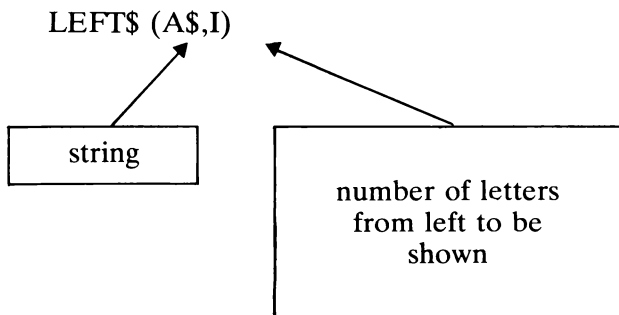
```
10 CLS
20 REPEAT
30 READ A$
40 IF A$ = "END-OF-DATA" THEN END
50 PROCdirection
60 PROCprint
70 UNTIL FALSE
80 DEF PROCdirection
90 direction = RND(9)
100 IF direction = 1 THEN X=1:Y=0
110 IF direction = 2 THEN X=1:Y=1
120 IF direction = 3 THEN X=0:Y=1
130 IF direction = 4 THEN X=-1:Y=1
140 IF direction = 5 THEN X=-1:Y=0
150 IF direction = 6 THEN X=-1:Y=-1
160 IF direction = 7 THEN X=0:Y=-1
170 IF direction = 8 THEN X=1:Y=-1
180 ENDPROC
190 DEF PROCprint
200 XPosition=RND(40):YPosition=RND(25)
210 PRINTTAB(XPosition,YPosition)LEFT$(A$,1)
220 FOR I=2 TO LEN(A$)
230 XPosition=XPosition+X:YPosition=YPosition+Y
240 PRINTTAB(XPosition,YPosition)MID$(A$,I,1)
250 NEXTI
260 ENDPROC
270 DATA DOORWAY, AT, WINDOW, CUP, CHAIR, BOOK
280 DATA END-OF-DATA
```

Did you notice two new commands here? They are LEFT\$ in line 210 and MID\$ in line 240. Here are some programs to show how these work.

```
10 A$="ROUNDAABOUT"  
20 FOR I=1 TO 10  
30 PRINT I,LEFT$(A$,I)  
40 NEXT I
```

>RUN

```
1          R  
2         RO  
3        ROU  
4       ROUN  
5      ROUN  
6     ROUNDA  
7    ROUNDA  
8   ROUNDA  
9  ROUNDA  
10 ROUNDA
```



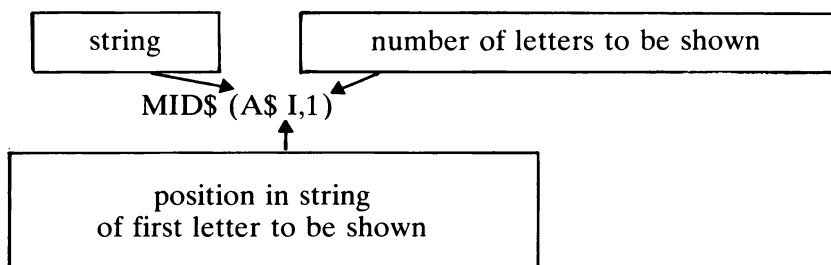
When I=1 then LEFT\$(A\$,I) will print one letter, starting at the left hand side of A\$. When I=2 then two letters are shown and so on.

Now change line 30 to read:

```
30 PRINT/I,MID$(A$,I,1)
```

```
>RUN
```

```
1      R
2      O
3      U
4      N
5      D
6      A
7      B
8      O
9      U
10     T
```



```
30 PRINT/I,MID$(A$,I,3)
```

```
>RUN
```

```
1      ROU
2      OUN
3      UND
4      NDA
5      DAB
6      ABO
7      BOU
8      OUT
9      UT
10     T
```

There is a third command used in printing parts of strings.

```
30 PRINT I,RIGHT$(A$,I)
```

```
>RUN
```

```
1      T
2      UT
3      OUT
4      BOUT
5      ABOUT
6      DABOUT
7      NDABOUT
8      UNDAABOUT
9      OUNDAABOUT
10     ROUNDAABOUT
```

RIGHT\$(A\$,I)

string

number of letters to be shown
from right hand side of string

Now try using LEN to allow words of any length to be INPUT.

```
10 INPUT "Input a word",A$
20 FOR I=1 TO LEN(A$)
30 PRINT I,RIGHT$(A$,I)
40 NEXT I
```

```
>RUN
```

```
Input a word?ABRACADABRA
```

```
1      A
2      RA
3      BRA
4      ABRA
5      DABRA
6      ADABRA
7      CADABRA
8      ACADABRA
9      RACADABRA
10     BRACADABRA
11     ABRACADABRA
```


We can extend the program to make a pattern.

```
10 INPUT "Input a word",A$
20 IF LEN(A$)>20 THEN PRINT "TOO LONG":GOTO 10
30 FOR I=1 TO LEN(A$)
40 PRINT LEFT$(A$,I);TAB(39-I)RIGHT$(A$,I)
50 NEXT I
```

For example:

```
Input a word? LLANFAIRPWLLGWYNGYLLGOGERYCH
WYRNDROBWLWLLANTISILIOGOGOGOCH
```

TOO LONG

```
>RUN
Input a word? FORMALDEHYDE
F                                     E
FO                                 DE
FOR                             YDE
FORM                          HYDE
FORMA                      EHYDE
FORMAL                    DEHYDE
FORMALD                LDEHYDE
FORMALDE              ALDEHYDE
FORMALDEH          MALDEHYDE
FORMALDEHY        RMALDEHYDE
FORMALDEHYD      ORMALDEHYDE
FORMALDEHYDE    FORMALDEHYDE
```

Can you find any more ways to make patterns from words?

Here is another way to use TAB:

```
10 CLS
20 TIME=0
30 REPEAT
40 sec=INT(TIME/100)
50 IF sec>59 THEN sec=sec-60:GOTO 50
60 min=INT(TIME/6000)
70 PRINTTAB(15,5)min;" : ";sec;" "
80 UNTIL FALSE
```

Your computer has a built-in clock. It doesn't count in hours, minutes and seconds, but in hundredths of a second. The clock can be zeroed, as in line 20 but the time has to be calculated from hundredths of a second, and this is done in lines 40, 50 and 60. In line 70, the time is repeatedly printed in the same place so that the display appears to change on the screen.

Here is another clock program that you will find useful with games, experiments and sports events.

```

10 DIM sec(15),min(15)
20 CLS:event=0:flag=0
30 PRINTTAB(11,2)"Multi-event Timer"
40 PRINTTAB(6,7)"Press any key to sta
rt clock"
50 start%=GET$
60 PRINTTAB(6,2)"Press any key to tim
e event"
70 TIME=0
80 REPEAT
90 sec=TIME/100
100 IF sec>59.9 THEN sec=sec-60:GOTO100
110 min=INT(TIME/6000)
120 PRINTTAB(15,5)min;"    ";INT(sec);"
"
130 PRINTTAB(0,7)"*****
*****"
140 PROCinstant
150 IF flag=1 THEN PROCevent
160 flag=0
170 UNTIL FALSE
180 DEF PROCinstant
190 instant=INKEY(0)
200 IF instant>0 THEN flag=1
210 *FX15
220 ENDPROC
230 DEF PROCevent
240 event=event+1
250 IF event>15 THEN PRINTTAB(5,23)"LI
ST FULL":END
260 sec(event)=sec

```

```
270 min(event)=min
280 PRINTTAB(4,event+7)"Event No. ";ev
ent;TAB(20);min(event);" : ";sec(event)
290 ENDPROC
```

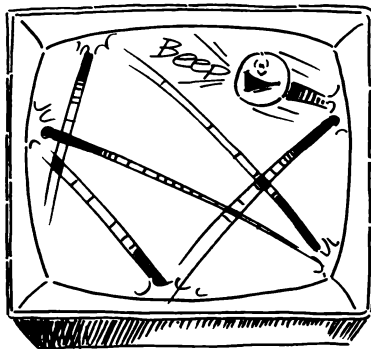
There are several new things here which need explaining. The main program loop is from lines 80 to 170. Line 50 starts the clock at the required moment. Each time around the loop, PROCinstant is called in line 140.

PROCinstant examines the keyboard to see if any key is being pressed (line 190). If a key *is* being pressed then the variable **flag** is set to 1. This is the same as raising a flag, so that when the program leaves PROCinstant, it can still tell whether a key was pressed. In line 210, *FX15 clears the keyboard buffer in case extra characters have been entered.

Back in the main loop at line 150, *if* flag=1 then PROCevent is called. PROCevent prints the event number, and the time (near enough) at which the key was pressed. In line 160, flag is set to 0, ready to be raised again when necessary.

This program allows you to store up to fifteen different event times. More event times *could* be stored, but more attention would have to be paid to the display because of scrolling.

Finally in this chapter we'll try a game – “Beeping Ball”.



(Notice that **integer variables** have been used to help speed it up.)

```

10 REM *** BEEPING BALL ***
20 REM * BY STEPHEN MILAN *
30 INPUT "DIFFICULTY 1-9 ",difficulty
40 MODE7:Xdirection%=1:Ydirection%=1:XX
=10:Y%=10:bat%=18
50 *FX11,5
60 REPEAT
70 FOR I=1 TO 10-difficulty:PROCmove:PR
OCbat:NEXT
80 PROCball
90 UNTIL FALSE
100 REM*****
110 DEF PROCmove
120 batdir$=INKEY$(0)
130 IF batdir$="Z" AND bat%>0 THEN bat%=
bat%-1
140 IF batdir$="X" AND bat%<33 THEN bat%=
bat%+1
150 *FX15,1
160 ENDPROC
170 DEF PROCbat
180 PRINTTAB(bat%,23);" ____ "
190 ENDPROC
200 REM*****
210 DEF PROCball
220 time=TIME:REPEAT:UNTIL TIME>time+5
230 PRINTTAB(XX,Y%);" "
240 XX=XX-Xdirection%:Y%=Y%-Ydirection%
250 IF XX<1 OR XX>38 THEN Xdirection%=-X
direction%
260 IF Y%<1 OR (Y%=23 AND XX>bat%-1 AND
XX<bat%+5) THEN Ydirection%=-Ydirection%:Y
DU7
270 IF Y%=24 THEN GOTO
280 PRINTTAB(XX,Y%);"o"
290 ENDPROC
300 *FX11,50
310 VDU7:END

```

Quite a few new ideas here.

The main program loop is between lines 50 and 100, calling just three PROCEDURES.

Line 30 invites the player to select how difficult the game is to be. This is controlled in line 80, by altering the number of times that PROCbat is called relative to PROCmove and PROCball.

Line 50 alters the speed of the **keyboard auto-repeat** so that there is no delay, and this is reset in line 310. Incidentally, if you **[ESCAPE]** from this program before it gets to line 310, then you will find it impossible to type properly because the computer will auto-repeat every letter no matter how fast you tap the keys. You will need to **[BREAK]** to reset the auto-repeat delay (remembering to type OLD if you want to keep the program).

PROCmove checks to see if keys "Z" or "X" have been pressed to move the "bat" to the left or the right. If so, it alters the value of bat% accordingly.

PROCbat prints the "bat" at TAB (bat%). Note that the "bat" has a space at each end within the string. This is so that when it moves, it erases the edge of the old bat.

PROCball prints the "ball". Line 220 is a short delay to slow the ball down. Line 230 prints a space at the old ball position to erase the previous ball. Line 240 calculates the new ball position. Lines 250 and 260 check that the ball is still within the court and if not, alter the direction. VDU7 makes a beep. In line 270, if the ball gets past the bat, then the program is ended (after resetting the auto-repeat delay).

There are lots of ways that this program can be extended and improved. Try printing instructions on the screen so that new players will know which buttons to press. And what about some kind of scoring, perhaps how many times the ball hits the bat?

WORD SQUARE 2

30 words which have been introduced in Chapters 5 to 8 are packed into this wordsquare.

R	O	N	U	T	F	G	C	J	K	N	P
T	O	S	R	T	L	O	O	P	E	P	H
I	E	N	E	M	H	E	M	A	R	K	U
M	I	C	A	P	H	A	E	L	M	I	N
E	L	G	D	A	A	P	S	H	I	F	T
N	N	N	E	L	O	R	G	R	I	N	I
N	E	N	I	T	U	O	R	B	U	S	L
E	S	C	T	H	S	C	R	A	I	S	T
O	T	F	M	U	N	E	X	T	Y	N	T
P	E	I	B	H	E	D	I	M	R	R	A
L	D	A	T	A	R	U	I	U	C	H	E
A	L	E	N	R	D	R	T	S	T	E	P
M	O	D	N	A	R	E	S	C	A	P	E
F	O	R	E	L	R	I	G	H	T	L	R
I	P	O	T	C	R	A	S	H	T	A	S
C	S	O	R	E	S	T	O	R	E	T	B

ARRAY
CRASH
DATA
DIM
ENDPROC

LEN
LOOP
MID
NESTEDLOOPS
NEXT

RESTORE
RETURN
RIGHT
SHIFT
STEP

ESCAPE
FOR
GET
GOSUB
LEFT

PROCEDURE
RANDOM
READ
REMARK
REPEAT

SUBROUTINE
TAB
TIME
TO
UNTIL

9 Design For Good Programming

By now we've met most of the programming features that will be covered in this book. The rest of the chapters are concerned with screen display, colour, sound and high resolution graphics – all things to help us to make the **OUTPUT** from our programs more attractive. But now that we have a good idea what programming is about, we should start to think about the qualities that go into a *good* program.

Let's think about the **qualities** that a program should have. Here is a list of seven (you may be able to think of more).

A program should be:

Accurate

Reliable

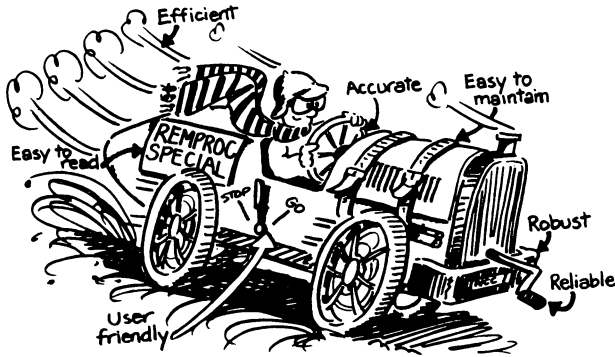
Robust (it shouldn't crash because of bugs or unexpected **INPUT**)

Efficient (it shouldn't be too slow, or use up too much memory space)

User-friendly (it should be easy for the user to use, have adequate instructions etc)

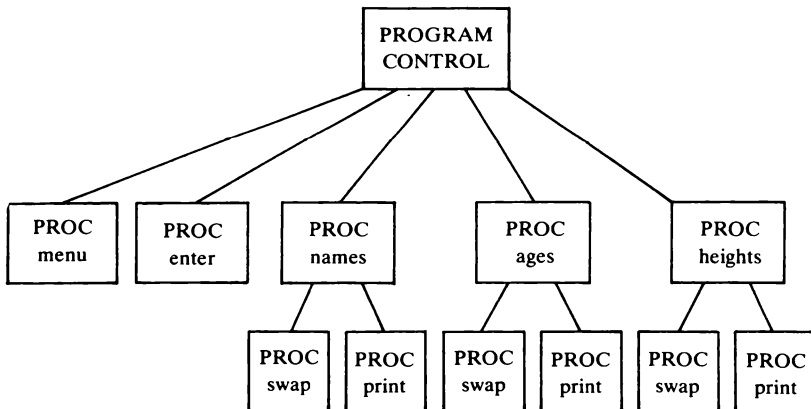
Easy to maintain (you'll want to change the program from time to time, so this should be easy to do)

Easy to read (programs often need to be read by humans as well as computers, so use **REMs**, procedures and meaningful variable names)



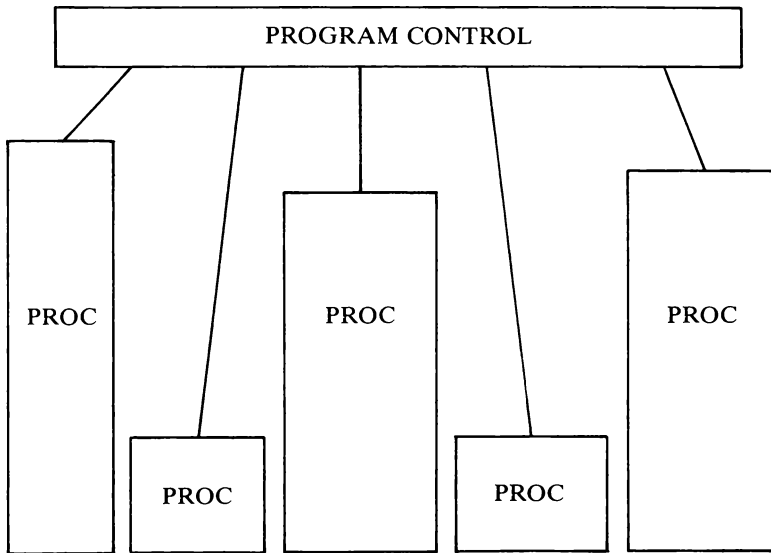
Once you get an idea for a program there is a great temptation to sit down at the keyboard and start to key in straight away. Professional programmers see programming in two different stages: *program design* and *program coding* (keying in). In the **program design** stage you should carefully consider what it is that you want the program to do, and how it should be constructed. No builder (in his right mind) would design a house by piling bricks on top of each other!

Starting with an overall view of the program is known as a **top down** approach. Starting with the main part of the program (which will control the rest), you can decide what PROCEDURES you will need. So the structure will look something like this (this is LISTSORT from Chapter 7):



The main part of the program calls up the procedures which perform the various tasks which need to be done. These procedures themselves may call up other procedures which perform other tasks. Once the structure is decided on, it is then easier to write each procedure to do the task required of it.

The other way of designing a program is known as the **bottom up** approach. In this method the programmer starts by writing each procedure, and then in the end writes a control section to join them all together. The trouble is that not having an overall plan, none of the procedures are likely to fit the others very well, and the control section may be difficult to write and not work very well. It would be a bit like this:



Here all the procedures are likely to be rather higgledy piggledy and difficult to fit together. So although **bottom up** is the approach that many people start by using, it isn't the best.

Although **top down** is the "right" approach to programming, many good programmers use a method which is somewhere between **top down** and **bottom up**. You *should* have an overall plan for

the program before you start coding, but as soon as you start to work on the procedures you will probably think of ways to improve the program as a whole. Don't be afraid to go back and change the overall plan. The detailed programming should only be done with the overall plan in mind, but there is nothing to say that the overall plan can't be improved.

A lot of the fun in writing your own computer programs is being able to improve them and get that "little bit extra" out of them. Structured programming makes this so much easier.

10 A La Mode

So far in this book nothing has been said about the different MODEs that are available on the BBC microcomputer (except for a brief mention in line 20 of “Beeping Ball” in the last chapter). There are eight MODEs available on the BBC Model B. The Model A, which has less memory space, has four of these. The MODEs control the number of characters that are displayed on each line across the screen, how many colours can be shown at one time and how fine is the detail of the graphics. There is a choice because the more characters, colour and graphics detail shown on the screen, the more memory is needed to store and display these. If a lot of memory is used for the display, then less is left for the program and data.

	characters across screen	lines down screen	colours	high resolution graphics	memory space used
(model A and B)					
MODE 7	40	25	16*	NO	1K
MODE 6	40	25	2	NO	8K
MODE 5	20	32	4	YES	10K
MODE 4	40	32	2	YES	10K
(model B only)					
MODE 3	80	25	2	NO	16K
MODE 2	20	32	16*	YES	20K
MODE 1	40	32	4	YES	20K
MODE 0	80	32	2	YES	20K

* In fact, the choice of colours is not as good as it looks. There are really only eight colours, the other eight are the same colours but

flashing. Of the eight steady colours, one of these is black and the other white. The colours are:

steady		flashing
0	black	8
1	red	9
2	green	10
3	yellow	11
4	blue	12
5	magenta	13
6	cyan	14
7	white	15

The letter K, when referring to computers, is a measure of the memory capacity that a computer has. The size of the memory is measured in **bytes**, each byte can store one character. One byte isn't very much and we normally talk in **kilobytes**. A kilobyte is actually 2^{10} , or 1024 bytes. We can easily think of it as *about* one thousand bytes. The Model A BBC computer has 16K bytes of Random Access Memory (RAM) that can be used to store program, data and screen information. The BBC Model B has 32 K of RAM. As you can see from the list, MODEs 0 to 3 need more memory space for the screen display than there is in the Model A, and so are not available in that Model.

MODE 7 uses up far less memory than any of the other modes, and although we can use colour, there are no user-definable graphics (coming later in this chapter) or high resolution graphics (coming in Chapter 12) available in this mode. However, not all programs need these features and MODE 7 is a very useful mode.

Try this:

```
10 FOR I=7 TO 4 STEP-1
20 MODE I
30 PRINTTAB(5,5)"This is mode ";I
40 FOR J=48 TO 125
50 PRINT CHR$(J)
60 NEXT J
70 TIME=0:REPEAT UNTIL TIME=500
80 NEXT I
```

If you have a Model B, then line 10 should read:

```
10 FOR I=7 TO 0 STEP-1
```

We can select a different mode simply by keying in a direct command, eg:

```
MODE 5
```

or else it can be incorporated into a program statement line (but *not* during a PROCEDURE). If the mode is changed during a program though, this will always clear the screen and may do funny things to the program. If the program and data use a lot of memory space, then some of this memory may be taken over by the screen display and the previous information lost. Be warned!

If you have a Model B machine you will see that two of the MODEs, 0 and 3, display 80 letters as characters across the screen. This is very useful for some purposes, when you want to see a lot of information at any one time. But these letters are really too small to be seen on an ordinary TV set, even a black and white one. They are meant to be seen on a TV **monitor** which is a special TV set. These can't usually receive pictures from an aerial. They give a very clear picture from a computer, but cost a lot more than an ordinary TV set. In time, more television set makers will bring out combined TV receivers/monitors for use with computers and video-cassette and video-disk players

In the program that we just ran we used a new keyword, CHR\$ (sometimes pronounced "character string" or "character dollar"). In the MODEs *other* than MODE 7, the ones that can normally be displayed on the screen range from CHR\$32 (a blank space!) to CHR\$127 (which stands for DELETE).

Try this:

```
10 CLS
20 ch = 31
30 FOR I = 1 TO 4
40 FOR J = 0 TO 23
50 ch = ch + 1
```

```

60 PRINTTAB(I*7,J);ch;" ";CHR$(ch)
70 NEXT J
80 NEXT I

```

RUN this program and it will display the characters and their CHR\$ numbers. Try the program in MODE 4.

With the BBC computer it is also possible to draw your own characters. You can design up to 32 of them (in fact with a lot of effort, you *could* redefine the entire character set). The characters that we can easily define for ourselves are CHR\$224 to CHR\$255.

Try this:

```

10 MODE 4
20 VDU23,224,255,129,129,129,129,129
129,255
30 PRINTTAB(5,5)CHR$224
40 FOR I=1 TO 10
50 PRINTTAB(10+I)CHR$224;
60 NEXT I
70 FOR J=1 TO 10
80 PRINTTAB(10,10+J)CHR$224
90 NEXT J
100 END

```

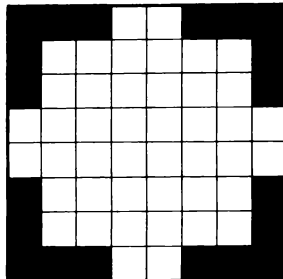
Try this change to “Beeping Ball”:

```

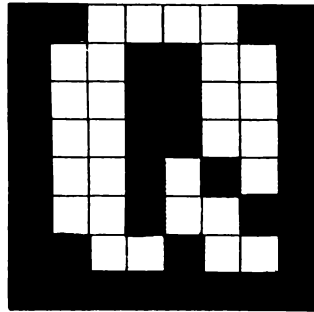
25 VDU23,224,24,126,126,255,255,126,126,24
280 PRINTTAB(X%,Y%);CHR$224

```

and the result will be a solid ball.

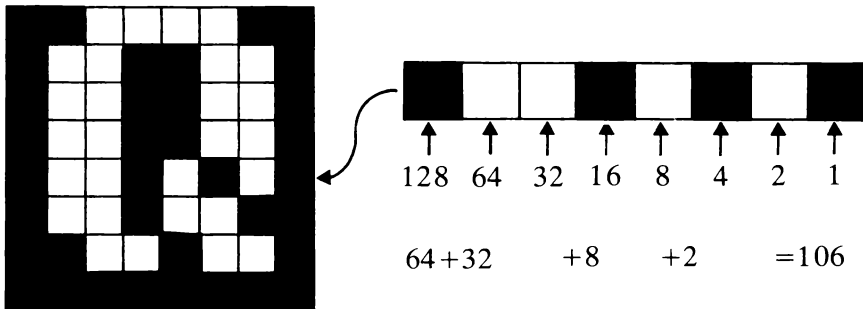


This is the way that it is done. Every character on the screen is made up of 64 dots, 8 rows of 8, like this:



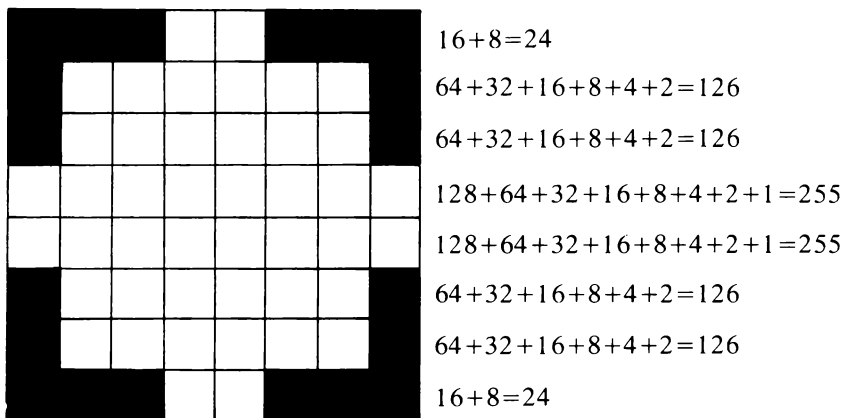
This is a capital Q

Some dots will be white and some black to make up the shape of the character. Each **line** of dots is held in the computer's memory as a number, and the number will depend on how the dots are distributed along the line.

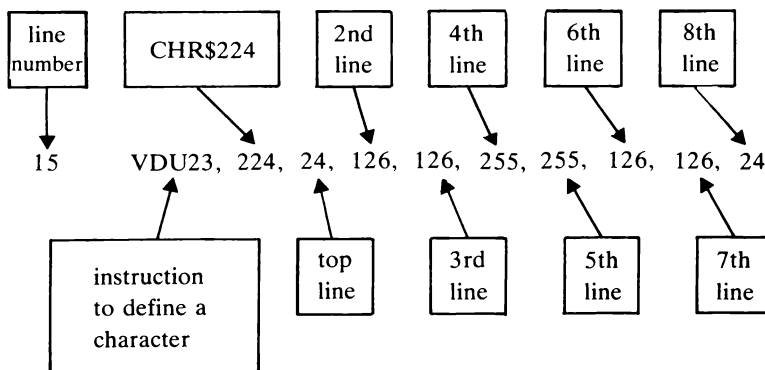


The dot at the right-hand end of the line has the value 1, the next to the left 2, the next 4, the next 8 and so on, until the dot at the left-hand end of the line has the value of 128.

The values of the dots that you want to show are added together. For the ball it was like this:



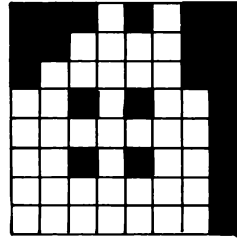
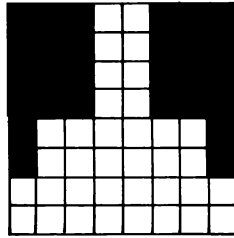
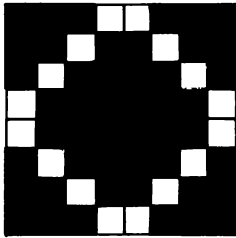
and the instructions to the computer made up like this:



In MODE 5 (also MODE 2 on Model B) there are only 20 characters shown across the screen. This makes the text difficult to read, but the graphics (particularly characters that you define yourself) can look very good indeed. These modes also have more colours available at any one time than the others, four colours in MODE 5 (and MODE 1 on Model B) and 16 colours in MODE 2 (although half the colours in MODE 2 are flashing, these can be combined with user-defined graphics to produce very striking effects).

Try making up some VDU23 statements for yourself.

Try these shapes:



Here is a program. Complete lines 10, 20 and 30 to define the above shapes. Then RUN the program to check the results.

```

10  VDU 23, 224, —, —, —, —, —, —, —, —,
20  VDU 23, 225, —, —, —, —, —, —, —, —,
30  VDU 23, 226, —, —, —, —, —, —, —, —,
40  MODE 5
50  FOR I = 1 TO 5
60  FOR J = 1 TO 5
70  P.TAB(I, J + 5); CHR$224
80  P.TAB(I + 8, J + 5); CHR$225
90  P.TAB(I + 14, J + 5); CHR$226
100 NEXT J: NEXT I

```

(The answers are at the end of the index at the back of the book, but don't cheat; try and work them out yourself.)

Try this:

```

10 REM***** WALK *****
20 MODE4
30 REM*** HIDE CURSOR ***
40 VDU23;11,0;0;0;0
50 VDU23;224,24,60,60,24,16,48,88,84
,23,225,0,0,0,0,0,0,0,0

```

```

60 VDU23,226,82,145,16,40,36,68,132,
71,23,227,4,6,15,15,6,8,6,14
70 VDU23,228,22,21,12,6,30,18,18,3,2
3,229,0,0,128,0,0,0,0,128
80 VDU23,230,3,7,7,3,1,1,1,3,23,233,
1,1,1,1,1,3,2,3
90 VDU23,232,192,128,0,128,128,128,1
28,192,23,231,0,128,128,0,0,0,0,0
100 VDU23,234,0,1,1,0,0,0,0,0,23,235,
192,224,224,192,64,96,224,80
110 VDU23,236,0,0,0,0,0,0,1,1,23,237,
128,216,192,96,112,224,128,128
120 one$=CHR$225+CHR$224+CHR$225+CHR$
8+CHR$8+CHR$8+CHR$10+CHR$225+CHR$226+CH
R$225
130 two$=CHR$225+CHR$227+CHR$225+CHR$
8+CHR$8+CHR$10+CHR$228+CHR$229
140 three$=CHR$225+CHR$230+CHR$231+CH
R$8+CHR$8+CHR$10+CHR$233+CHR$232
150 four$=CHR$225+CHR$234+CHR$235+CHR
$8+CHR$8+CHR$10+CHR$236+CHR$237
160 C=1
170 REPEAT
180 PRINTTAB(C,10);one$;
190 PROCDELAY
200 PRINTTAB(C,10);two$;
210 PROCDELAY
220 PRINTTAB(C,10);three$;
230 PROCDELAY
240 PRINTTAB(C,10);four$;
250 PROCDELAY
260 C=C+1
270 UNTIL C=36
280 PRINTTAB(0,20)
290 REM*** RESET CURSOR ***
300 VDU23;11,255;0;0;0
310 END
320 DEFPROCDELAY
330 FOR I=1 TO 200:NEXT
340 ENDPROC

```

In this program the characters have been defined to show the various positions of a running man. These are then printed together with CHR\$8 (cursor backspace) and CHR\$10 (cursor down) so that they are printed as blocks of four gradually moving across the screen.

If you want to see the user-defined shapes then add this line:

```
285 FOR I = 224 TO 236:P.CHR$(I);" ";I:NEXT
```

Now we have a program called "Shoot the Rapids". This is a game which uses user-defined graphics. These are defined in line 260 and 270.

```

10 REM ** SHOOT THE RAPIDS **
20 REM ** BY STEPHEN MILAN **
30 MODE 4
40 VDU19,0,4,0,0,0:VDU19,1,2,0,0,0
50 PROCinstructions
60 REPEAT
70 PROCsetUp
80 TIME=0
90 REPEAT
100 PROCrun
110 UNTIL character<>32
120 time=TIME
130 PROCclearup
140 UNTIL D$<>"Y" AND D$<>"y"
150 END
160 DEFPROCinstructions
170 PRINTTAB(12,4);"SHOOT THE RAPIDS"
,,
180 PRINT"In this game you are in the
small boat at the bottom of the screen
travelling down a river surrounded by
jungle.""
190 PRINT"You steer the boat by Press
ing the ""
200 PRINT" <Z> key to go LEFT and the
""
210 PRINT" <M> key to go RIGHT.""

```

```

220 PRINT "If you hit the bank the cro
codiles get you."
230 PROCdifficulty
240 ENDPROC
250 DEFPROCsetup
260 VDU23,224,8,28,42,119,42,8,8,0
270 VDU23,225,126,255,255,255,255,255
,255,126
280 REM ** ALTER AUTOREPEAT
290 *FX 11,1
300 across=20:left = 10:right =10
310 FOR I=1 TO 32
320 PRINTSTRING$(10,CHR$(224));STRING
$(20," ");STRING$(10,CHR$(224));
330 NEXT
340 ENDPROC
350 DEFPROCrun
360 PRINTTAB(across,30);CHR$(225);
370 REM ** CLEAR KEYBOARD
380 *FX 15,0
390 leftbank=left+RND(5)-3
400 IF leftbank<10 THEN leftbank=10
410 rightbank=right+RND(5)-3
420 IF rightbank<10 THEN rightbank=10

430 IF rightbank+leftbank>35 THEN 390

440 left=leftbank:right=rightbank
450 PRINTTAB(0,0);CHR$(11);STRING$(le
ft,CHR$(224));STRING$(40-left-right," ")
;STRING$(right,CHR$(224));
460 FOR I=1 TO delay:NEXT:REM ** DELAY

470 D$=INKEY$(0)
480 IF D$="Z" THEN across = across -1
490 IF D$="M" THEN across = across +1
500 PRINTTAB(across,30);
510 AX=135:character=(USR(&FFF4)AND&F
F00)/%100
520 ENDPROC

```

```

530 DEFPROCcleanup
540 FOR I=1 TO 5000:NEXT I:REM ** DELAY
550 REM ** CLEAR KEYBOARD
560 *FX15,0
570 CLS
580 PRINTTAB(0,9);"You lasted ";INT(time/100);" seconds"
590 PRINTTAB(9,15);"DO YOU WANT ANOTHER GO?"
600 REM ** RESET AUTOREPEAT
610 *FX 11,50
620 D#=GET#
630 IF D#="Y" OR D#="y" THEN PROCdifficulty
640 ENDPROC
650 DEF PROCdifficulty
660 *FX15,1
670 PRINT"" HOW FAST DO YOU WANT TO GO? (1-9)"
680 difficulty#=GET#
690 IF VAL(difficulty#)<1 OR VAL(difficulty#)>9 THEN PRINT"INPUT A NUMBER BETWEEN 1 AND 9":GOTO 680
700 delay=100-10*VAL(difficulty#)
710 ENDPROC

```

If you have a Model B machine, or a Model A machine with 32K of memory, then you can increase the number of colours on the screen by altering these lines:

```

30 MODE1
40 VDU19,0,4,0,0,0:VDU19,1,1,0,0,0:VDU19,2,2,0,0,0
55 COLOUR2
60 COLOUR1:PRINTTAB(across,30);CHR$(225):COLOUR2

```

If you have a Model B machine *and* BBC joysticks you can also alter these lines:

```

460 delay=ADVAL(2)/600:FOR I=1 TO delay:NEXT I
470 IF ADVAL(1)<25000 THEN across = across -1
480 IF ADVAL(1)>35000 THEN across = across +1

```

How does it work? The program is controlled by two nested REPEAT. . .UNTIL loops (lines 60 to 140 and 90 to 110).

The busiest part of the program is in PROCrun from lines 350 to 520. This randomly sets up and prints a new line of jungle (lines 390 to 450). There is a delay in line 460 to control the speed of the program. Line 450 shows how to make the screen scroll downwards (the opposite way to usual). PRINTTAB(0,0) sends the cursor to the top left-hand corner of the screen and CHR\$(11) (which is "cursor up") sends everything on the screen down one line.

In line 470 we have a feature that we haven't come across before – INKEY\$. INKEY\$ is a bit like GET\$ except that INKEY\$ doesn't have to wait for something to be entered, it simply looks to see if anything has been entered. (It *can* be made to wait for a certain time before moving on, by altering the number in brackets). Anything that has been entered is checked in lines 480 and 490. If it is "Z" then **across** (which is the position of the boat across the screen) is decreased so that the boat will move to the **left**. If INKEY\$ was "M" then across is increased so that the boat will move to the **right**.

In lines 500 and 510, the position on the screen where the boat will be is examined to see if it is river or jungle. If it is river then it will be CHR\$(32) – a blank space. If it is jungle then it will be CHR\$(224). The value is stored in **character**.

PROCrun then ends and the program goes back to line 80. If **character** = 32 (that is, the boat *hasn't* hit the jungle) then the program stays in the REPEAT. . .UNTIL loop and goes through PROCrun for every new line of jungle that is printed. **Try improving the program by counting these to give a score.**

User-Defined Keys

As well as being able to define your own characters, you can define your own keys. These are the ten red keys at the top of the keyboard, marked f0 to f9.

Press f1. Probably nothing will happen. Now enter:

*KEY1 THIS IS KEY 1 RETURN

Now press f1 and

THIS IS KEY 1

will appear on the screen.

So far so good, but not very useful.

Here is a program that will define your keys to save you some effort when entering or changing programs. You will need to enter (or LOAD) the program you want to work on. The key will stay defined even if you type NEW or press BREAK (but not if you switch the computer off!)

```

10 REM**PROGRAM TO DEFINE RED KEYS**
20 *KEY0LOAD""!M
30 *KEY1RUN!M
40 *KEY2LIST
50 *KEY3PRINT
60 *KEY4TAB(
70 *KEY5FOR I=1 TO
80 *KEY6NEXT
90 *KEY7REPEAT
100 *KEY8UNTIL
110 *KEY9MODE7!M
120 CLS:PRINTTAB(4,10)"KEYS DEFINED -
LOAD NEXT PROGRAM"
130 END

```

The symbol! (which appears on the screen as `!!` in MODE 7) followed by M means that the command is automatically RETURNed.

It's also useful to temporarily define keys with long variable names if they are to be used often in a program. This saves a lot of typing (and mistakes) or the use of COPY

In later versions of the BBC computer, pressing the SHIFT key together with one of the red keys will produce the CHR\$s for different colours in TELETEXT mode.

Cut off the strip at the side of the page at the back of the book and place it under the clear plastic above the red keys on your computer. This will then show how the keys have been defined.

The other side has been left blank for you to fill in when you write your own key-defining program.

(If you don't want to cut the book, copy out the strip on a separate piece of paper.)

11 A Colourful Character

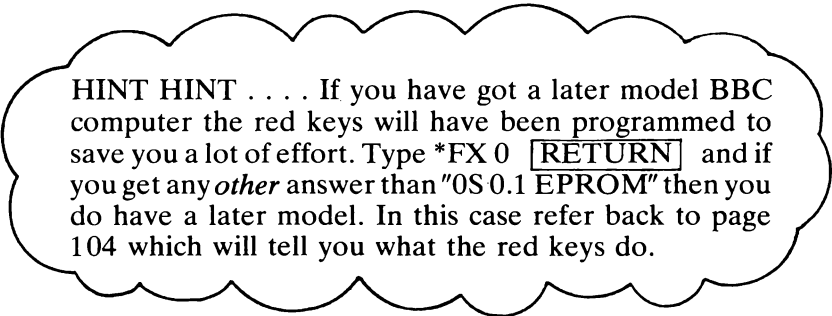
In order to see the results of the colour commands, it will be necessary to connect the computer to a colour TV set. Of course, you can enter the programs using a black and white set, and even RUN them. This way you can do all the keying, and then SAVE the programs on cassette until you have access to a colour TV set.

There are two ways of producing colours on the BBC computer. One way is to use the TELETEXT method in MODE 7 and the other is to use the COLOUR command in the other modes.

TELETEXT COLOURS

MODE 7 gives a TELETEXT display, similar to that used on CEEFAX, ORACLE and PRESTEL. This doesn't need much memory and although the colours are easy to use, the method is a bit tedious and possibilities limited. The instructions for the colours are contained in the PRINT statements.

Try this (don't forget to use COPY to save typing on lines that are similar):



HINT HINT If you have got a later model BBC computer the red keys will have been programmed to save you a lot of effort. Type *FX 0 RETURN and if you get any *other* answer than "0S 0.1 EPROM" then you do have a later model. In this case refer back to page 104 which will tell you what the red keys do.

```

10 MODE7
20 PRINTTAB(7,5);"THIS IS A TELETEXT
DISPLAY"
30 PRINTTAB(0,8)"Teletext allows the
use of these colours"
40 PRINTTAB(5);"RED";TAB(25);"GREEN"
,
50 PRINTTAB(5);"YELLOW";TAB(25);"BLU
E"
60 PRINTTAB(5);"MAGENTA";TAB(25);"CY
AN"
70 PRINTTAB(13,18);"also flashing"
80 PRINT"";"RED";"GREEN";"YELLOW";"BL
UE";"MAGENTA";"CYAN";"WHITE"

```

RUN this and you will see that all the text is printed in white.

Now, using COPY , alter line 40 to read:

```
40PRINTTAB(5);"RED";TAB(25);CHR$130;"GREEN"
```

When you RUN this now, "GREEN" is printed in **green**. So putting CHR\$ 130 into the line means that the characters which follow are printed in green. But the following lines are all printed in white, so the effect only works on one line. You will also see that "GREEN" has moved over to the right one space, so even though the **control character** isn't printed on the screen, it still takes up a space.

Now alter line 50 to read:

```
50PRINTTAB(5);CHR$131;"YELLOW";TAB(25);"BLUE"
```

Now if you RUN the program you will see that both "YELLOW" *and* "BLUE" are in yellow. So unless a control character is altered, its effect lasts for all of the rest of the line, even through separate strings.

So here are lines 40 to 80, showing the correct colours:

```

40 PRINTTAB(5);CHR$129;"RED";TAB(25)
;CHR$130;"GREEN"

```

```

50 PRINTTAB(5);CHR$131;"YELLOW";TAB(
25)CHR$132;"BLUE"
60 PRINTTAB(5);CHR$133;"MAGENTA";TAB
(25);CHR$134;"CYAN"
70 PRINTTAB(13,18);"also";CHR$136;"f
lashing"
80 PRINT""CHR$129;"RED";CHR$130;"GRE
EN";CHR$131;"YELLOW";CHR$132;"BLUE";CHR
$133;"MAGENTA";CHR$134;"CYAN";CHR$135;"
WHITE"

```

Run this to check that all is correct.

Now alter lines 70 and 80 to read:

```

70 PRINTTAB(13,18);"also";CHR$136;"f
lashing"
80 PRINT""CHR$136CHR$129;"RED";CHR$1
30;"GREEN";CHR$131;"YELLOW";CHR$132;"BL
UE";CHR$133;"MAGENTA";CHR$134;"CYAN";CH
R$135;"WHITE"

```

and add this new line:

```

90 PRINTTAB(12)CHR$136;"flashing";CH
R$137;"steady"

```

So you can see that CHR\$ 136 produces flashing letters, while CHR\$ 137 restores a steady display.

Have you noticed that where we added the extra control characters in line 80, "WHITE" has now gone over the end of the line and the last two letters no longer flash?

TELETEXT also allows the use of double height characters.

Alter line 20 to read:

```

20 PRINTTAB(7,5);CHR$141;"THIS IS A
TELETEXT DISPLAY"

```

and add line 25.

```
25 PRINTTAB(7,6);CHR$(141);"THIS IS A  
TELETEXT DISPLAY"
```

RUN this and you will see that the heading is in larger letters. These *have* to be printed on two lines (although you don't have to use TAB, so long as the letters line up). These double height characters can be coloured in the usual way and don't have to be the same colour top and bottom.

Try this:

```
20 PRINTTAB(7,5);CHR$(141);CHR$(131);"TH  
IS IS A TELETEXT DISPLAY"  
25 PRINTTAB(7,6);CHR$(141);CHR$(133);"TH  
IS IS A TELETEXT DISPLAY"
```

Ugh!

Finally, we can have a coloured background on different lines.

To change the colour of the background we have to use three CHR\$s. First we select a colour as we would do for a character. If we want red, we would use CHR\$ 129. Then we would have to use CHR\$ 157 which gives us a **new background** and then we have to change the colour of the characters, or else we won't see them against the background.

So if we wanted yellow letters on a red background we'd use (press CONTROL and L to clear the screen and try this direct command):

```
PRINT CHR$(129)CHR$(157)CHR$(131);"Yellow on red"
```

Here is the complete program with lines 40, 50 and 60 altered to give coloured backgrounds:

```
10 MODE7  
20 PRINTTAB(7,5);CHR$(141);CHR$(131);"TH  
IS IS A TELETEXT DISPLAY"  
25 PRINTTAB(7,6);CHR$(141);CHR$(133);"TH  
IS IS A TELETEXT DISPLAY"
```

```
30 PRINTTAB(0,8)"Teletext allows the
   use of these colours"
40 PRINTTAB(5,10);CHR$129;"RED";TAB(
25);CHR$130;"GREEN"
50 PRINTTAB(5,12);CHR$131;"YELLOW";T
AB(25)CHR$132;;"BLUE"
60 PRINTTAB(5,14);CHR$133;"MAGENTA";
TAB(25);CHR$134;"CYAN"
70 PRINTTAB(13,18);"also";CHR$136;"f
lashing"
80 PRINT' 'CHR$136CHR$129;"RED";CHR$1
30;"GREEN";CHR$131;"YELLOW";CHR$132;"BL
UE";CHR$133;"MAGENTA";CHR$134;"CYAN";CH
R$135;"WHITE"
90 PRINTTAB(12)CHR$136;"flashing";CH
R$137;"steady"
```

Here is a list of the main control characters and their effect:

CHR\$ 129 = red
CHR\$ 130 = green
CHR\$ 131 = yellow
CHR\$ 132 = blue
CHR\$ 133 = magenta
CHR\$ 134 = cyan
CHR\$ 135 = white*
CHR\$ 136 = flash
CHR\$ 137 = steady*
CHR\$ 140 = normal height*
CHR\$ 141 = double height
CHR\$ 156 = black background*
CHR\$ 157 = new background

(* every new line automatically starts with these settings unless altered by the program.)

Here is a classical "Hangman" program which uses MODE 7 for text, graphics and colour:

```

10 REM *** HANGMAN ***
20 MODE 7
30 PROCstartup
40 REPEAT
50 flag=0:check$=""
60 PROCfindword
70 REPEAT
80 *FX15
90 correct=0
100 PRINTTAB(0,5);CHR$131;"Input your l
etter ";CHR$255
110 letter$=GET$
120 IF ASC(letter$)<65 OR ASC(letter$
)>90 THEN VDU7:GOTO 100
130 FOR I=1 TO LEN(check$)
140 IF letter$=MID$(check$,I,1) THEN
VDU7:I=26:NEXT:GOTO 100
150 NEXT I
160 check$=check$+letter$
170 PRINTTAB(19,5);CHR$135;letter$
180 FOR I=1 TO LEN(word$)
190 IF letter$=MID$(word$,I,1) THEN P
RINTTAB(I-1,10);letter$:letter=letter+1
:correct=1
200 NEXT I
210 TIME=0:REPEAT UNTIL TIME=100
220 PRINTTAB(20,5);"    "
230 PROCused
240 IF letter=LEN(word$) THEN PROCgot
250 IF flag=1 THEN 50
260 IF correct=0 THEN error=error+1:P
ROCvictim
270 UNTIL error=8
280 PRINTTAB(0,3);CHR$129CHR$157CHR$1
31;"    SORRY YOU ARE DEAD!!"
290 PRINT;"THE WORD WAS:-    "
300 PRINTTAB(0,10);word$

```

```
310 PROCanother90
320 UNTIL FALSE
330 DEF PROCanother90
340 PRINTTAB(0,23);CHR$131CHR$157CHR$
132CHR$136;"DO YOU WANT ANOTHER GO ?"CHR
R$137"(Y/N)"
350 YN$=GET$
360 CLS
370 IF YN$<>"Y" AND YN$<>"y" THEN PRI
NTTAB(14,12)"Good bye":END
380 *FX15,0
390 ENDPROC
400 DEF PROCstartup
410 PRINTTAB(12,7)CHR$129CHR$141"HANG
MAN"
420 PRINTTAB(12,8)CHR$129CHR$141"HANG
MAN"
430 TIME=0
440 numberofwords=0:letter=0:error=0
450 REPEAT
460 READ word$
470 numberofwords=numberofwords+1
480 UNTIL word$="NOMOREWORDS":numbero
fwords=numberofwords-1
490 DIM usedwords(numberofwords)
500 RESTORE
510 PRINT' 'CHR$131;"There are ";numbe
rofwords;" words in this game"
520 PRINT'CHR$131"You are only allowe
d 8 mistakes"
530 PRINT' 'CHR$131"Press the"CHR$134"
<SPACE>"CHR$131"bar to continue"
540 wait$=GET$
550 FOR I=1 TO numberofwords:usedword
s(I)=0:NEXT
560 ENDPROC
570 DEF PROCfindword
580 letter=0:error=0:correct=0
590 word=RND(numberofwords)
600 IF usedwords(word)=1 THEN 590
```



```

610 usedwords(word)=1
620 FOR I=1 TO word
630 READ word#
640 NEXT
650 RESTORE
660 CLS
670 PRINTTAB(7,0);CHR#134;"ABCDEFGH IJ
KLMNOPQRSTUVWXYZ"
680 PRINTTAB(0,10);
690 FOR I=1 TO LEN(word#)
700 PRINT"_";
710 NEXT
720 PRINTTAB(21);"    "
730 ENDPROC
740 DEF PROCused
750 PRINTTAB(8+(ASC(letter#)-65),1);"
^"
760 ENDPROC
770 DEF PROCgot
780 PRINTTAB(0,4);CHR#129CHR#136CHR#1
57CHR#135;"        YOU GOT IT ! "
790 PRINTTAB(0,5)"
    "

800 PROCanother90
810 flag=1
820 ENDPROC
830 DEF PROCvictim
840 ON error GOTO 860,880,920,940,980
,1030,1070,1090
850 ENDPROC
860 PRINTTAB(15,21);CHR#133;STRING$(2
0,CHR#255)
870 ENDPROC
880 FOR I=1 TO 14
890 PRINTTAB(32,6+I);CHR#133CHR#255
900 NEXT
910 ENDPROC
920 PRINTTAB(17,7);CHR#133;STRING$(17
,CHR#255)
930 ENDPROC

```

```

940 FOR I=1 TO 3
950 PRINTTAB(28+I,7+I);CHR#133;"\"
960 NEXT
970 ENDPROC
980 PRINTTAB(18,8);"!"
990 PRINTTAB(16,9);CHR#131;"@@@"
1000 PRINTTAB(15,10);CHR#133;"('.')"
1010 PRINTTAB(17,11);" _ "
1020 ENDPROC
1030 FOR I=1 TO 3
1040 PRINTTAB(16,12+I);CHR#129;STRING#
(3,CHR#255);CHR#133
1050 NEXT I
1060 ENDPROC
1070 PRINTTAB(12,12);CHR#129;"[";STRIN
G(9,CHR#255);"]";CHR#133
1080 ENDPROC
1090 FOR I=1 TO 3
1100 PRINTTAB(16,15+I);CHR#132CHR#255;
" ";CHR#255CHR#133
1110 NEXT
1120 PRINTTAB(14,19);CHR#129"000 000";
CHR#133
1130 ENDPROC
1140 DATACOMPUTER,KEYBOARD,ARRAY,PROCE
DURE
1150 DATARANDOM,REPEAT,DISC DRIVE,SOUND
1160 DATACASSETTE,AEROPLANE,DUSTBIN,FI
NISH
1170 DATALEAP,BOOKSHELF,PUZZLE,CHESS
1180 DATANOMOREWORDS

```

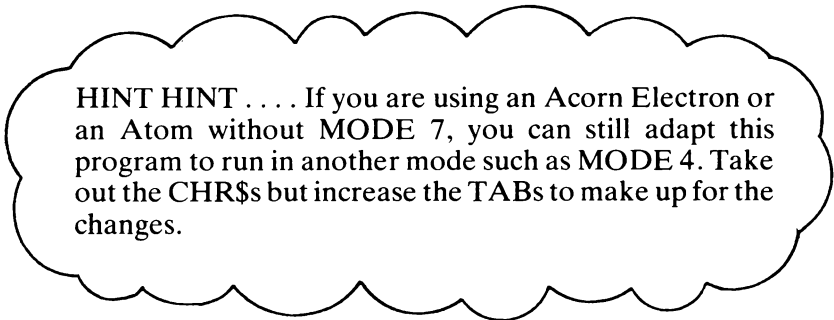
As this program is in MODE 7, there is plenty of space left in the memory to add new words. As it stands, the program uses about 2.5K bytes of memory space, the screen display another 1K byte leaving about 12.5K bytes left for words! Enough for about 1000 words, even on a Model A machine. Extra words can be added in DATA statements with line numbers *less* than the line which reads DATANOMOREWORDS. Change the line number of DATANOMOREWORDS so that it is always the last line of the

program. Be sure to **DELETE** any earlier ones or else the program won't be able to get past them. Be careful with the words in **DATA** statements, don't leave spaces between the words and the commas (,) and be very careful not to leave spaces at the end of lines before pressing **RETURN** .

There are some useful ideas in the program for when *you* write your own programs.

Line 120 checks that the character entered is a capital letter between A and Z. Lines 130 to 150 compare the letter just entered to all the accepted letters (in the string **check\$**) that have already been entered. Any letters which have previously been entered are rejected. Line 160 adds the latest accepted letter to **check\$**. Line 190 compares the entered letter to all the letters in the word to be guessed.

PROCstartup (lines 400 to 560) counts the number of words, declares the variable and sets up an array **word\$(I)** which checks whether a word has already been used in this game. If so, it is rejected and another is chosen.



HINT HINT . . . If you are using an Acorn Electron or an Atom without **MODE 7**, you can still adapt this program to run in another mode such as **MODE 4**. Take out the **CHR\$**s but increase the **TAB**s to make up for the changes.

COLOURS IN MODES 6 TO 0

MODES 0, 3, 4 and 6 can display two colours at once.

MODES 1 and 5 can display four colours at once.

MODE 2 can display 16 colours at once.

Normally when switched on, the two-colour modes display black and white. The four-colour modes display black, white, red and

yellow. MODE 2 can display the same colours as MODE 7. The two and four-colour modes can be made to display other colours than those mentioned, although the maximum number of colours that can be shown at any one time cannot be increased.

Type in (as direct commands):

MODE 5

COLOUR 1 RETURN – the text on the next line turns red.

COLOUR 2 RETURN – the text on the next line turns yellow.

COLOUR 3 RETURN – the text on the next line returns to white.

Now try COLOUR 0 RETURN and try typing something. You can see the cursor moving, but no text. What happened?

COLOUR 0 is black and so the text is there, but you can't see it because it has merged with the background. Press RETURN and the computer will probably answer with "MISTAKE" although you won't be able to see it. Type the following very carefully:

COLOUR 129RETURN – the background on the next line turns red (now we can see the black letters).

COLOUR 130RETURN – the background on the next line turns yellow.

COLOUR 131RETURN – the background on the next line turns white.

COLOUR 129: COLOUR 1 RETURN – white letters on a red background.

COLOUR 128: COLOUR 1 RETURN – white on black as usual.

Here is a list of the colour numbers in the four-colour modes (MODES 5 and 1):

text		background
0	black	128
1	red	129
2	yellow	130
3	white	131

Here are the colours in two-colour modes (MODES 6, 4, 3 and 0):

text		background
0	black	128
1	white	129

These numbers refer to the **logical colours**.

Key in this program and RUN it to see the effects.

```

10 MODE 5
20 FOR textcolour =0 TO 3
30 FOR backgroundcolour=131 TO 128 S
TEP-1
40 COLOUR textcolour
50 COLOUR backgroundcolour
60 PRINTtextcolour;" * ";backgroundc
colour
70 NEXT backgroundcolour
80 NEXT textcolour

```

If you have a Model B machine, or a Model A with 32K then alter these lines:

```

10 MODE 2
20 FOR textcolour=0 TO 7
30 FOR backgroundcolour=135 TO 128 S
TEP-1
35 FOR I=1 TO 500:NEXT

```

and then for an even more spectacular effect alter these lines again.

```

20 FOR textcolour=0 TO 15
30 FOR backgroundcolour=143 TO 128 S
TEP-1

```

These programs will demonstrate all the possible combinations of **logical** background and foreground colours.

As mentioned previously it *is* possible to change the colours *actually* shown on the screen in the two and four-colour modes.

Here is a list of the **actual** colours and their numbers:

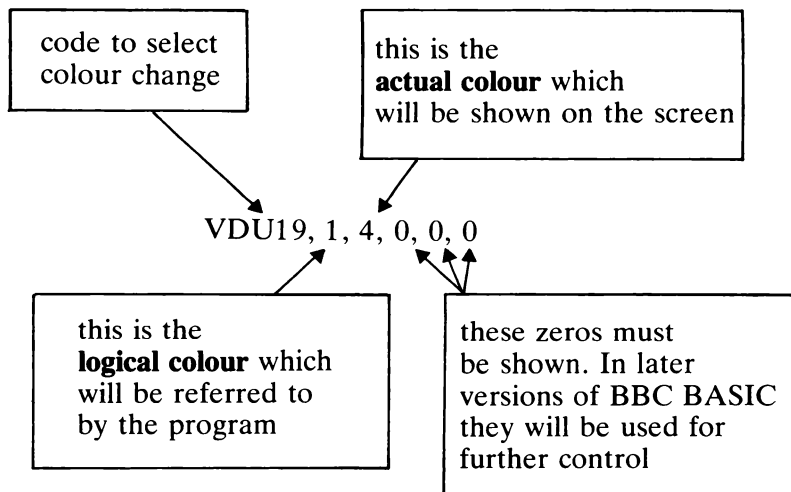
0	black
1	red
2	green
3	yellow
4	blue
5	magenta
6	cyan
7	white
8	flashing black/white
9	flashing red/cyan
10	flashing green/magenta
11	flashing yellow/blue
12	flashing blue/yellow
13	flashing magenta/green
14	flashing cyan/red
15	flashing white/black

The change is made by using a VDU19 command, like this: suppose that in a two-colour mode you want **blue** letters on a **cyan** background. We would need these commands:

```
10  MODE 4
20  VDU 19, 1, 4, 0, 0, 0
30  VDU 19, 0, 6, 0, 0, 0
```

RUN this and type something and you will see that you are typing in blue letters on a cyan background.

How can you use the VDU19 command to select the colours that *you* want?



The **logical** colours and **actual** colours can refer to background colours as well (as in line 30), but don't use the logical background numbers (128 and above) as these don't work here. Instead, use the logical foreground number for the same colour.

Here is a program to demonstrate all the combinations of coloured foreground and background that are possible in just a two-colour mode:

```

10 MODE 4
20 COLOUR1:COLOUR128
30 FOR backgroundcolour=15 TO 0 STEP
-1
40 FOR textcolour=15 TO 0 STEP-1
50 VDU19,1,textcolour,0,0,0
60 VDU19,0,backgroundcolour,0,0,0
70 PRINT"****";textcolour;"**";backg
roundcolour;"*****"
80 NEXT textcolour
90 NEXT backgroundcolour
100 VDU20
110 PRINT"****7**0*****"
```

If you want to examine each combination for longer, change line 50 to read:

```
50 wait$ = GET$
```

Each combination will then stay on the screen until a key is pressed. The bottom line will tell the numbers of the actual foreground and background colours. From time to time all the text on the screen will disappear. This is because the foreground and background are printed in the same colour. In fact this happens with the first combination.

Here are some effective text colour combinations in two-colour modes. Try them and fill in the colours you got in the space provided.

VDU19, 0, 4, 0, 0, 0 (try this in MODE 6 also).

— — — — —

VDU19, 0, 7, 0, 0, 0, 19, 1, 0, 0, 0, 0

(Notice how you can put two VDU commands into the same statement – make sure that you put in the right number of zeros.)

— — — — —

VDU19, 0, 1, 0, 0, 0, 19, 1, 3, 0, 0, 0

— — — — —

Here is a “Maths Quiz” program:

```
10 REM***** MATHS QUIZ *****
20 MODE4
30 CLS
40 VDU23,224,0,16,0,124,0,16,0,0
50 VDU23,255,255,255,255,255,255,255
,255,255
60 VDU19,0,7,0,0,0
70 VDU19,1,0,0,0,0
80 wrongtotal=0:righttotal=0:ch=0
90 REPEAT
100 CLS
110 PRINTTAB(15,5);"MATHS TEST"
120 PRINTTAB(0,10);"Which level (1-9)"
```



```

"
130 level$=GET$
140 level=VAL(level$)
150 IF level<1 OR level>9 THEN VDU7:G
OTO 100
160 go=0
170 REPEAT
180 CLS
190 PROCchoose
200 PROCproblem
210 PROCanswer
220 go=go+1
230 UNTIL go=10
240 PROCresults
250 PRINTTAB(5,20)"Do you want anothe
r go?(Y/N)"
260 anothergo$=GET$
270 IFanothergo$<>"Y" AND anothergo$<
>"y" THEN END
280 UNTIL FALSE
1000 DEF PROCchoose
1010 sign=RND(4)
1020 IF sign=1 THEN sign$="+":colour=1
1030 IF sign=2 THEN sign$="-":colour=4

1040 IF sign=3 THEN sign$="*":colour=2
1050 IF sign=4 THEN sign$="/":colour=5
1060 VDU19,1,colour,0,0,0
1070 ENDPROC
1500 REPEAT
2000 DEF PROCproblem
2010 wrong=0
2020 vtab=5:htab=20:PROCsign
2030 A=RND(10*level)
2040 B=RND(10*level)
2050 IF sign$="/" THEN PROCdivision
2060 PRINTTAB(10,15);A;
2070 IF sign$="*" THEN PRINT" x ";:GOT
O 2100

```

```

2080 IF sign$="/" THEN PRINT " ";CHR$(2
24);" ";GOTO2100
2090 PRINT " ";sign$;" ";
2100 PRINT;B;" = ";
2110 ENDPROC
3000 DEF PROCanswer
3010 PRINTTAB(25,15);"          "
3020 PRINTTAB(25,15);:INPUTanswer$
3030 coranswer$="A"+sign$+"B"
3040 IF VAL(answer$)<<(EVAL(coranswer$
+.01) AND VAL(answer$)>>(EVAL(coranswer$
)-.01) THEN PROCright ELSE PROCwrong
3050 IF flag=1 THEN 3010
3060 IF flag=2 THEN ENDPROC
4000 DEFPROCresults
4010 CLS
4020 PRINTTAB(5,10);"You got ";rightto
tal;" correct answers"
4030 PRINTTAB(8);"and ";wrongtotal;"
wrong ones"
4040 ENDPROC
5000 DEFPROCsign
5010 IF sign$="+" THEN PROCplus
5020 IF sign$="-" THEN PROCminus
5030 IF sign$="*" THEN PROCTimes
5040 IF sign$="/" THEN PROCdivide
5050 ENDPROC
6000 DEFPROCdivision
6010 C=A*B:Z=A:A=C:C=Z
6020 ENDPROC
7000 DEF PROCright
7010 FORI=1 TO 20
7020 VDU19,0,RND(7),0,0,0
7030 VDU19,1,RND(7),0,0,0
7040 FOR delay=1 TO 150:NEXT
7050 NEXT
7060 VDU19,0,7,0,0,0
7070 VDU19,1,colour,0,0,0
7080 flag=2
7090 righttotal=righttotal+1

```

```
7100 ENDPROC
8000 DEF PROCwrong
8010 PRINTTAB(0,18);"Sorry that is not
the correct answer"
8020 wrong=wrong+1:flag=1
8030 IF wrong=3 THEN PRINT"The correct
answer is:- ";EVAL(conanswer$):wrong=0
8040 FOR delay=1 TO 4000:NEXT
8050 PRINTTAB(0,18);SPC(39)
8060 PRINTTAB(0,19);SPC(39)
8070 wrongtotal=wrongtotal+1
8080 ENDPROC
9000 DEFPROCplus
9010 FOR I=-2 TO 2
9020 PRINTTAB(htab,vtab+I);CHR$255
9030 PRINTTAB(htab+I,vtab);CHR$255
9040 NEXT
9050 ENDPROC
10000 DEFPROCminus
10010 FOR I=-2 TO 2
10020 PRINTTAB(htab+I,vtab);CHR$255
10030 NEXT
10040 ENDPROC
11000 DEFPROCTimes
11010 FOR I=-2 TO 2
11020 PRINTTAB(htab-I,vtab+I);CHR$255
11030 PRINTTAB(htab+I,vtab+I);CHR$255
11040 NEXT
11050 ENDPROC
12000 DEFPROCdivide
12010 FOR I=-2 TO 2
12020 PRINTTAB(htab+I,vtab);CHR$255
12030 NEXT
12040 FOR I=-2 TO 2 STEP 4
12050 PRINTTAB(htab,vtab+I);CHR$255
12060 NEXT
12070 ENDPROC
```

Maths Quiz uses MODE 4 which is a two-colour mode, but the colours are redefined for each type of question. There is also a multi-coloured display for a correct answer.

The main part of the program is in two nested REPEAT. . . UNTIL loops (lines 90 to 280 and 170 to 230). The inner loop calls the PROCEDURES as required, the outer one keeps the program going until the player decides that he or she doesn't want another go.

There are one or two new things to look at.

Line 3040 uses EVAL (which is short for EVALuate). This works out the answer to a mathematical expression which is entered as a string. EVAL is useful here because it has been easy to make up the question into a string using randomly chosen elements, PRINT the string and work out the answer.

Lines 8050 and 8060 use SPC(39). This is simply a compact way of telling the computer to print 39 blank spaces. Of course, any other number or a calculation could be put inside the brackets.

In PROCdivision (*called* in line 2050 and *shown* starting at line 6000) it was found best to start off a division by knowing the answer and then working out the question. The normal way always ended up with an answer like 7.7777777. In PROCdivision the two numbers are multiplied together ($C=A*B$). Then C and A are swapped around. Then A (which was C) is divided by B and the answer is always a sensible whole number.

You will certainly be able to improve this program. Try putting it into MODE 5 and use four colours at once (you'll need to change all the TABs). When you've read the next chapter, you will be able to add some sounds for correct and wrong answers.

12 Play It Again, Computer



Your BBC computer can make sounds. You can hear some every time you switch on, or SAVE a program. We've already used sound in our programs with VDU7.

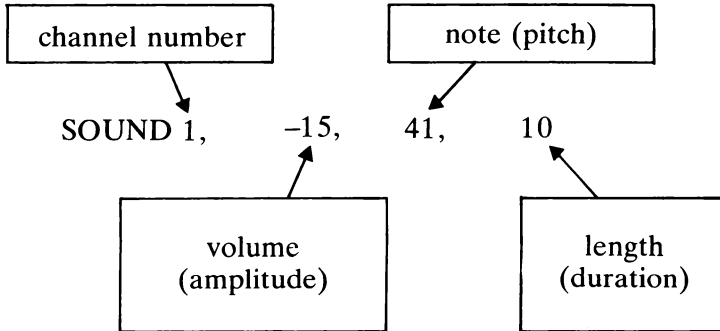
Type: VDU7 RETURN

There are two sound commands, SOUND and ENVELOPE. We'll try SOUND first.

Try this:

SOUND 1, -15, 41, 10 RETURN

This command contains four numbers (**parameters**) to control the sound of the note.



CHANNEL

There are four channels, 0 to 3. 0 makes noises rather than musical sounds. The other three channels can be sounded together to make chords.

AMPLITUDE

This controls the volume of the sound. -15 (surprisingly) is the loudest, 0 is the quietest. Numbers between 1 and 4 can also be used here, but only in connection with the ENVELOPE statement which we will try later.

Try this:

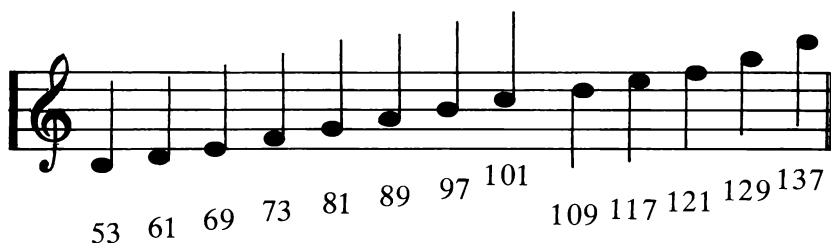
```

10 CLS
20 FOR volume=0 TO 15 STEP 1
30 PRINTTAB(10,10); " "; volume; " "
40 SOUND 1,volume,149,20
50 I=INKEY(200)
60 NEXT volume
70 SOUND 1,0,0,0
  
```

PITCH

This determines the actual note that is sounded, whether it is high or low. The scale goes from 0 to 255. For the musical, semitones are four apart. Middle C is 53.

To increase the note by a semitone, the pitch number is increased by 4, for a whole tone, 8.



Try this:

```

10 CLS
20 FOR Pitch=5 TO 253 STEP 4
30 PRINTTAB(10,10);"  ";Pitch;"  "
40 SOUND 1,-15,pitch,20
50 I=INKEY(200)
60 NEXT Pitch
70 SOUND 1,0,0,0

```

DURATION

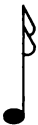
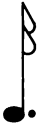



This controls the length of time for which the note lasts. It is measured in 1/20ths of a second.





```

10 CLS
20 FOR duration=1 TO 20
30 PRINTTAB(10,10);"  ";duration;"  "
40 SOUND 1,-15,149,duration
50 I=INKEY(200)
60 NEXT duration
70 SOUND 1,0,0,0

```

Again, for musical readers, here are the approximate note equivalents:

				
semi- quaver	dotted semi quaver	quaver	dotted quaver	crochet
2	3	4	6	8

			
dotted crochet	minim	dotted minim	semi- breve
12	16	24	32

Try this (use the COPY key to save effort):

```

10 SOUND1,-15,129,10
20 SOUND1,-15,117,10
30 SOUND1,-15,129,10
40 SOUND1,-15,117,10
50 SOUND1,-15,101,10
60 SOUND1,-15,109,5
70 SOUND1,-15,117,5
80 SOUND1,-15,121,10
90 SOUND1,-15,109,10
100 SOUND1,-15,129,10
110 SOUND1,-15,117,10
120 SOUND1,-15,101,20

```

RUN this and you will hear a familiar tune.

Here is the same tune, the program is written in a different way.

```

10 REPEAT
20 READ amplitude,Pitch,duration
30 IF amplitude=0 THEN END
40 SOUND1,amplitude,Pitch,duration
50 UNTIL FALSE
500 DATA-15,129,10,-15,117,10,-15,129
,10
510 DATA-15,117,10,-15,101,10,-15,109
,5
520 DATA-15,117,5,-15,121,10,-15,109,
10
530 DATA-15,129,10,-15,117,10,-15,101
,20,0,0,0

```

RUN this and then try these changes:

```
5 ENVELOPE 1,1,0,0,0,0,0,0,127,-2,-10,-1,126,0
```

```
40 SOUND 1, 1, pitch, duration
```

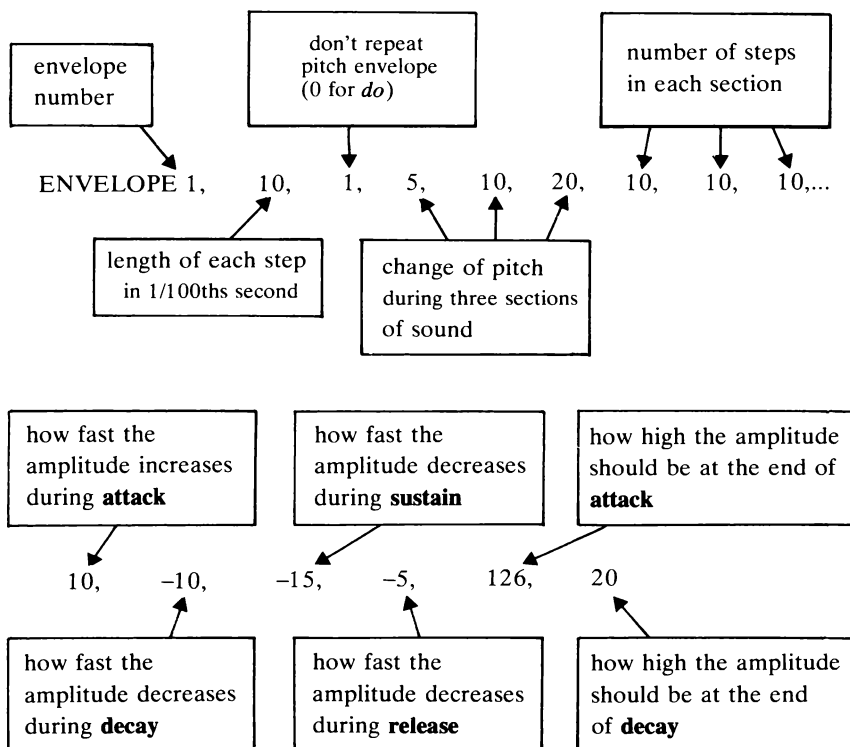
If you want the program to RUN without stopping then change:

```
30 IF amplitude = 0 THEN RESTORE
```

(you will need to press **ESCAPE** to stop the program).

ENVELOPE is a powerful and complex command. We can only start to experiment with it here, but we can hear some of its effects.

ENVELOPE is in two parts, the **pitch** envelope and the **amplitude** envelope. We can define four different envelopes, and each will have fourteen parameters.



Here is a program which uses ENVELOPE to change the amplitude envelope:

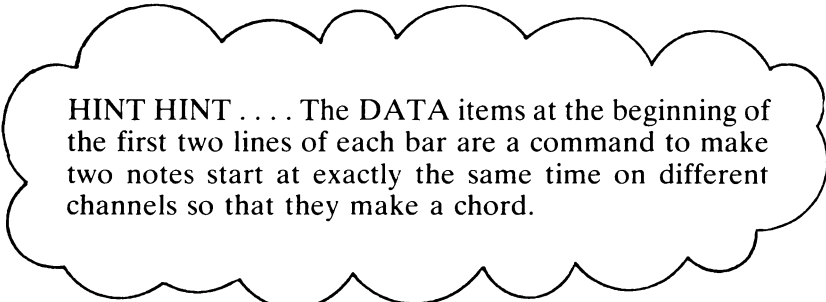
```
10 REM ***** ANTHEM *****
20 ENVELOPE 1,1,0,0,0,0,0,0,127,-2,-10,-1,
126,0
30 RESTORE
40 REPEAT
50 READ channel,amplitude,pitch,duration
60 IF channel=0 THEN END
70 SOUND channel,1,pitch,duration
80 UNTIL FALSE
90 REM ***** BAR No.1 *****
100 DATA%0101,-15,137,10
110 DATA%0102,-10,89,24
120 DATA1,-12,137,6
130 DATA1,-12,145,8
140 REM ***** BAR No.2 *****
150 DATA%0101,-15,133,12
160 DATA%0102,-10,21,24
170 DATA1,-12,137,4
180 DATA1,-12,145,8
190 REM ***** BAR No.3 *****
200 DATA%0101,-15,105,8
210 DATA%0102,-10,89,24
220 DATA1,-12,105,8
230 DATA1,-12,109,8
240 REM ***** BAR No.4 *****
250 DATA%0101,-15,105,12
260 DATA%0102,-10,89,24
270 DATA1,-12,145,4
280 DATA1,-12,137,8
290 REM ***** BAR No.5 *****
300 DATA%0101,-15,145,8
310 DATA%0102,-10,21,24
320 DATA1,-12,137,8
330 DATA1,-12,133,8
340 REM ***** BAR No.6 *****
350 DATA%0101,-15,137,24
360 DATA%0102,-10,89,24
```

```
370 REM ***** BAR No.7 *****
380 DATA&0101,-15,117,8
390 DATA&0102,-10,89,24
400 DATA1,-12,117,8
410 DATA1,-12,117,8
420 REM ***** BAR No.8 *****
430 DATA&0101,-15,117,12
440 DATA&0102,-10,89,24
450 DATA1,-12,109,4
460 DATA1,-12,105,8
470 REM ***** BAR No.9 *****
480 DATA&0101,-15,109,8
490 DATA&0102,-10,21,24
500 DATA1,-12,109,8
510 DATA1,-12,109,8
520 REM ***** BAR No.10 *****
530 DATA&0101,-15,109,12
540 DATA&0102,-10,21,24
550 DATA1,-12,105,4
560 DATA1,-12,97,8
570 REM ***** BAR No.11 *****
580 DATA&0101,-15,105,8
590 DATA&0102,-10,89,24
600 DATA1,-12,109,4
610 DATA 1,-12,105,4
620 DATA 1,-12,97,4
630 DATA 1,-12,89,4
640 REM ***** BAR No.12 *****
650 DATA&0101,-15,105,12
660 DATA&0102,-10,89,24
670 DATA1,-12,109,4
680 DATA1,-12,117,8
690 REM ***** BAR No.13 *****
700 DATA&0102,-10,61,8
710 DATA&0101,-15,125,5
720 DATA 1,-12,109,3
730 DATA&0101,-12,105,8
740 DATA&0102,-10,69,8
750 DATA&0101,-12,97,8
760 DATA&0102,-10,21,8
```

```

770 REM ***** BAR No.14 *****
780 DATA&0101,-15,89,24
790 DATA&0102,-15,41,24
800 DATA0,0,0,0

```



HINT HINT . . . The DATA items at the beginning of the first two lines of each bar are a command to make two notes start at exactly the same time on different channels so that they make a chord.

Try changing the ENVELOPE command with these:

```

20 ENVELOPE 1,5,0,0,0,0,0,12,32,-25,-25,126,80
20 ENVELOPE 1,5,0,0,0,0,0,127,-32,-10,-25,126,0

```

Changing the pitch parameters can produce some surprising results. Try:

```

20 ENVELOPE 1,5,10,-10,-10,25,25,25,10,0,-5,126,100

```

Listen to the effect and try to see how the command works. Try experimenting yourself.

Finally in this chapter we have a program to use the computer's keyboard as a musical keyboard and display the notes played.

The program uses **high resolution graphics** which are not explained until the next chapter. Just type the program in and you will be able to understand later how it works.

```

10 REM ***** KEYBOARD *****
20 REM *** BY STEPHEN MILAN ***
30 MODE4
40 PROCdraw
50 REPEAT
60 FOR KEY%=1 TO 16
70 IF KEY%=1 THEN RESTORE:PRESSED%=0

```

```

      80 READ KEYNO%,KEY$
      90 IF INKEY(KEYNO%) THEN NEWPITCH%=I
,NSTR(note$,KEY$)*4+48:PRESSED%=1:VOLUME
%=-10
     100 IF NEWPITCH%<>PITCH% THEN on%=0:A
CROSS%=ACROSS%+64:PITCH%=NEWPITCH%:IF A
CROSS%>1200 THEN PROCclear
     110 SOUND&11,VOLUME%,PITCH%,2
     120 PROCstave
     130 NEXT
     140 IF PRESSED%=0 THEN VOLUME%=0:PITC
H%=0:NEWPITCH%=0
     150 UNTIL FALSE
     160 DEFPROCdraw
     170 MOVE0,600:MOVE0,300:PLOT85,1280,6
00:PLOT85,1280,300
     180 GCOLOR,0
     190 FORI=0 TO 1280 STEP 128
     200 MOVEI,300:DRAWI,600
     210 DRAWI+124,600:DRAWI+124,300
     220 DRAWI,300
     230 NEXT
     240 FORI=64 TO 1160 STEP 128
     250 IF I=64 OR I=576 OR I=960 THEN 30
0
     260 MOVEI,600
     270 MOVEI,450
     280 PLOT85,I+120,600
     290 PLOT85,I+120,450
     300 NEXT
     310 KEY$="ZXCVBNM,./"
     320 VDU5:FOR I=0 TO LEN(KEY$)
     330 MOVE128*I+48,400:PRINTMID$(KEY$,I
+1,1)
     340 NEXT
     350 KEY$=" DFG JK ;"
     360 GCOLOR,1
     370 VDU5:FORI=0 TO LEN(KEY$)
     380 MOVE128*I+112,550:PRINTMID$(KEY$,
I+1,1)

```

```

390 NEXT
400 VDU4
410 FORI=1 TO 5
420 MOVE0,700+I*32
430 DRAW1280,700+I*32
440 NEXT
450 PITCH%=0:VOLUME%=0:PRESSED%=0:ACR
OSS%=0
460 note$="ZXDCFVGBNUMK,.,/":NEWPITCH
%=0
470 on%=0
480 ENDPROC
490 DEF PROCstave
500 IF PRESSED%=0 OR on%=1 THEN ENDPR
OC
510 on%=1
520 HEIGHT%=KEY%
530 IF KEY%=11 THEN HEIGHT%=2
540 IF KEY%=12 THEN HEIGHT%=3
550 IF KEY%=13 THEN HEIGHT%=4
560 IF KEY%=14 THEN HEIGHT%=6
570 IF KEY%=15 THEN HEIGHT%=7
580 IF KEY%=16 THEN HEIGHT%=9
590 VDU5:MOVEACROSS%,732+HEIGHT%*16
600 IF KEY%>10 THEN PRINT"#o" ELSE PR
INT" o"
610 VDU4
620 MOVEACROSS%+56,720+HEIGHT%*16
630 DRAWACROSS%+56,780+HEIGHT%*16
640 ENDPROC
650 DEF PROCclear
660 FORI=0 TO 12:PRINTTAB(0,I);SPC(40
)
670 NEXT
680 FORI=1 TO 5
690 MOVE0,700+I*32
700 DRAW1280,700+I*32
710 NEXT
720 ACROSS%=0
730 ENDPROC

```

```
740 DATA-98,Z,-67,X,-83,C,-100,V
750 DATA-101,B,-86,N,-102,M,-103,","
760 DATA-104,.,-105,/, -51,D,-68,F
770 DATA-84,G,-70,J,-71,K,-88,;
```


13 It's All A Plot

You've already moved text around the screen using the TAB command. We've also defined our own characters using VDU 23. In these ways we've been able to have graphics output from our programs onto the screen.

The BBC computer has another way of drawing onto the screen. This is known as **high resolution graphics**.

Do you remember how TAB works? This moves the cursor from wherever it is on the screen, directly to the point where you want to start printing. So that it can do this, we give it two numbers (co-ordinates) to tell it where to go.

```
10 PRINT TAB (10,5)
```

The first number is the number of columns across the screen, and the second is the number of lines down. So in a 40-column, 32-line mode (MODE 4 for instance), TAB(20,16) would start printing more or less in the middle of the screen.

High resolution graphics (which works in MODES 0, 1, 2, 4 and 5) uses a similar method of moving the cursor. There is one important difference. Using TAB, the position TAB(0,0) is in the *top left-hand corner* of the screen (this is known as the **text origin**). Using high resolution graphics, the point with co-ordinates 0,0 (the **graphic origin**) is in the *bottom left-hand* corner.

Try this:

```
10  MODE 4
20  MOVE 100,100
```

```

30 DRAW 1180,100
40 DRAW 1180,924
50 DRAW 100,924
60 DRAW 100,100

```

RUN this and you should see a box drawn on the screen. There are two different statements, MOVE and DRAW.

```

20 MOVE 100,100

```

This line tells the graphics cursor to move from where it happens to be to the co-ordinates 100,100. In doing so it doesn't show on the screen.

```

30 DRAW 1180,100

```

tells the graphics cursor to draw a straight line from where it happens to be (in this case, 100,100) to the co-ordinates given, (1180,100).

Alter line 10 to read:

```

10 MODE 5

```

and RUN again.

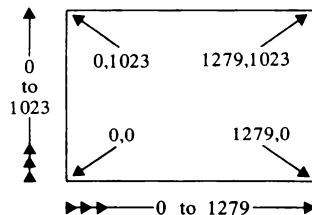
The result will be about the same as before, although the lines will be thicker. The co-ordinates used are the same throughout the five modes to be which support **hi-res** graphics, although the amount of detail shown will depend on the mode (and the amount of memory used). If you have a Model B machine, or a Model A with 32K you can also try:

```

10 MODE 0

```

The range of co-ordinates on the screen are:



Now try this:

```
10  MODE 4
20  A = 0: B = 0: C = 1279: D = 1023
30  INPUT "increment", inc
40  REPEAT
50  MOVE A,B
60  DRAW C,B
70  DRAW C,D
80  DRAW A,D
90  DRAW A,B
100  A = A+inc: B = B+inc: C = C-inc: D = D-inc
110  UNTIL B>D
```

So we can MOVE the cursor to a point and then DRAW a line from that point to another.

Now try changing these lines of the program:

```
65  D = D-inc
75  A = A+inc
85  B = B+inc
100  C = C-inc
```

Then try RUNning it several times. INPUT 10, 50 and 100 as increments. You'll see that as the increment gets bigger, so the angle of the line get bigger. Where the line is not quite horizontal or not quite vertical, then it is made up of broken straight lines. As the angle gets bigger, so the lines are made up of more and more small lines and the effect is less noticeable. Change line 10 to MODE 5 and see the effect emphasised.

Then add these lines:

```
15  FOR I = 1 TO 3
120 NEXT I
```

Then RUN again, using the same values for increment. Finally RUN again, using values of 100, 37 and 2.

Now we can try DRAWing figures that require more calculation.

```
10 REM ***** POLYGON *****
20 MODE 4
30 radius=500
40 INPUT "Number of sides ",N
50 MOVE 640,1024
60 FOR I=0 TO 360 STEP 360/N
70 DRAW SIN(RAD(I))*radius+640,
COS(RAD(I))*radius+512
80 NEXT I
```

The program works like this. N is the number of sides. This is used to find the number of degrees in each section (line 60). The algorithm which draws the shape is in line 70.

Try entering different values for N. Note that the more sides in the polygon, the more circular it becomes. This is how the computer draws circles. If the polygon has 72 or more sides then it cannot be distinguished from a circle.

In fact, MOVE and DRAW are similar to parts of a more powerful command called PLOT.

PLOT is followed by three numbers, the PLOT type and two co-ordinates.

PLOT 4,100,100 is the same as MOVE 100,100.

PLOT 5,1180,100 is the same as DRAW 1180,100.

Here is a program to demonstrate a new type of PLOT, PLOT 6. Key this in (note how the inner loop, lines 90 to 150, uses this PLOT twice on the same co-ordinates).

```
10 REM *** PLOT 6 DEMO ***
20 MODE 4
30 PROCdisc
40 PRINTTAB(0,0)"Press"'"<ESCAPE>"' "to
END"
50 REPEAT
60 PROCcross_hairs
70 UNTIL FALSE
80 REM *****
90 DEF PROCdisc
100 VDU29,640;512;
110 FOR angle=0 TO 360 STEP 10
120 MOVE0,0
130 DRAW SIN(RAD(angle))*500,COS(RAD(ang
le))*500
140 PLOT 85,SIN(RAD(angle+10))*500,COS(R
AD(angle+10))*500
150 NEXT angle
160 VDU29,0;0;
170 ENDPROC
180 DEF PROCdelay
190 FOR delay =1 TO 100:NEXT
200 ENDPROC
210 REM *****
220 DEF PROCcross_hairs
230 FOR I=1 TO 40
240 X=1279*I/40
250 Y=1023*I/40
260 FOR J=1 TO 2
270 MOVE X,0
280 PLOT 6,X,1023
290 MOVE 0,Y
300 PLOT 6,1279,Y
310 IF J=1 THEN PROCdelay
320 NEXT J
330 NEXT I
340 ENDPROC
```

This PLOT type can be very useful. It plots a line which at any point is the opposite colour of whatever was on the screen at that

point before. If you cross an area that was not in the background colour, then it will be printed in a contrasting colour, so that it can always be seen. Another use is for erasing a line that you have drawn. If you PLOT it over itself, then it completely disappears. This is used in the PLOTTER program that comes later, for drawing the cross-hairs which move about the screen. When they are first drawn they can be seen, no matter what they are over. Then when they are drawn for a second time they disappear and anything else that was touched is restored to what it was before.

Now try this:

```
10 MODE 4
20 FOR I=1 TO 20
30 PLOT 4,0,0
40 PLOT 4,RND(1279),RND(1023)
50 PLOT 1,100,0
60 PLOT 1,-50,100
70 PLOT 1,-50,-100
80 NEXT I
```

In this program using PLOT 1 we can DRAW to points **relative** to the one where the cursor is. It's just as if we change the graphics origin with every statement. So PLOT 4 is like MOVE and we use it to place the cursor randomly on the screen. PLOT 1 then plots a triangle **relative** to that point.

Try increasing the loop parameter in line 20 to 200.

Now to try combining colour with hi-res graphics. This needs the use of GCOL.

First in a two-colour mode, say MODE 4. As with COLOUR, two colours can be used at once, and these can be defined as any of the 16 colours available. In MODE 4 we can have one foreground colour and one background colour. If we don't give the computer any special instructions, then the background colour will be black and the foreground colour will be white.

Add these lines to the previous program:

```
110 GCOL 0,1
120 GCOL 1,3
```

and you will get yellow triangles on a red background next time that you RUN the program. This is because in line 110 we re-defined those areas normally black (0) as red (1) and in line 120, those areas normally white (1) were changed to yellow (3).

Now add these lines:

```

90  FOR I = 0 TO 15
100  A = I + 1: IF A = 16 THEN A = 0
110  GCOL 0, A
130  FOR delay = 1 TO 100: NEXT delay
140  NEXT I

```

Now to try in a four-colour mode. Here is a program to draw a histogram (a graph which shows quantities by columns):

```

10MODE5
20DIMname$(12),forecast(12),actual(12)
)
30PROCdata
40PROCcolours
50PROCerid
60PROCeraph
70REPEAT UNTIL FALSE
80DEF PROCcolours
90VDU19,0,6,0,0,0
100VDU19,1,0,0,0,0
110VDU19,2,1,0,0,0
120VDU19,3,4,0,0,0
130ENDPROC
140DEF PROCerid
150height=0
160FORI=1 TO counter-1
170IF forecast(I)>height THEN height=f
orcast(I)
180IF actual(I)>height THEN height=ac
tual(I)
190NEXT
200GCOLOR,1:MOVE100,100:DRAW100,1024
210MOVE100,100:DRAW1280,100

```



```
220FOR I=0 TO height
230MOVE100,INT(900/height)*I+100
240DRAW80,INT(900/height)*I+100
250NEXT
260FOR I=1 TO 12
270VDU5:MOVE99*I+100,125
280FOR J=1 TO 3
290PRINTCHR$(10);CHR$(8);MID$(name$(I
),J,1);
300NEXT
310NEXT
320VDU4
330ENDPROC
340DEF PROCgraph
350FOR I=1 TO counter-1
360GCOLOR.2
370MOVE99*I,100
380MOVE99*I+50,100
390PLOT85,99*I,INT(900/height)*forecast(I)+105
400PLOT85,99*I+50,INT(900/height)*forecast(I)+105
410GCOLOR.3
420MOVE99*I+25,100
430MOVE99*I+75,100
440PLOT85,99*I+25,INT(900/height)*actual(I)+105
450PLOT85,99*I+75,INT(900/height)*actual(I)+105
460NEXT
465GCOLOR.1:VDU5:MOVE0,100:PRINT"0"
466MOVE0,1024:PRINT;height
467MOVE640-(LEN(title$)/2)*64,1000:PRINTtitle$
468VDU4
470ENDPROC
480DEF PROCdata
485READtitle$
490counter=0
500REPEAT
```

```

510counter=counter+1
520READA$,B,C
530IF A$="*" THEN ENDPROC
540name$(counter)=A$
550forecast(counter)=B
560actual(counter)=C
570UNTIL FALSE
575DATAComputer sales
580DATAJAN,15,11
590DATAFEB,12,8
600DATAMAR,11,11
610DATAAPR,8,9
620DATAMAY,2,3
630DATAJUN,4,5
640DATAJUL,6,18
650DATAAUG,8,7
660DATASEP,8,8
670DATAOCT,7,8
680DATANOV,10,10
690DATADEC,3,15
700DATA*,0,0

```

This uses MODE 5 for four colours, but these are re-defined. You can re-define these yourself to any colour that you like by altering lines 90 to 120.

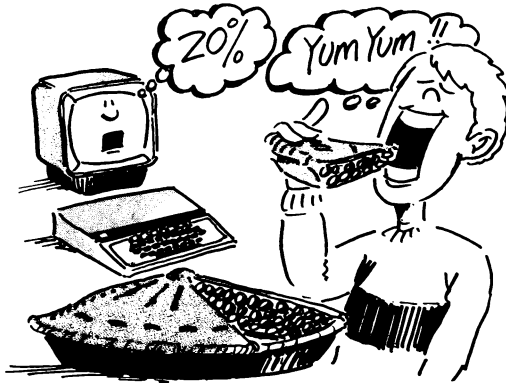
The values to be shown on the histogram are in the DATA statements. You can put in your own names and numbers here, or alter PROCdata so that you can enter them through INPUT statements at the start of the RUN, and then store them in an array. (You can get an idea how to do this from the next program – PIECHART.) You could (with a bit of programming effort) use this program to display the results of the LISTSORT program in Chapter 7.

PROCgrid draws the grid having first looked (in lines 170, 180) for the largest DATA item. PROCgrid then scales the grid to be just larger than the largest item. Try altering the DATA to see this work.

VDU 5 (in line 465) is a new command. This makes the text

cursor move to the same co-ordinates as PLOT. This means that you can position text accurately onto your graphics displays, without having to calculate from graphics co-ordinates to TAB.

The next program also displays data onto the screen so that it can be more easily understood. This time the data is turned into a PIECHART. That is where the various data items are shown as slices of a pie.



```

10 REM***** PIECHART *****
20 MODE5
30 DIM item_value(20),item_name$(20)
40 PROCdata
50 PROCcircle
60 PROCpiechart
70 END
80 DEFPROCpiechart
90 VDU19,2,2,0,0,0
100 VDU19,3,7,0,0,0
110 VDU29,640,512,
120 howmuch=0:colour=1
130 FOR I=1 TO item_number
140 colour=colour+1:IF colour=3 THEN c
colour=1
150 GCOLOR,colour
160 FOR J=0 TO item_value(I)-1

```

```

170 MOVE 0,0
180 MOVE SIN(RAD((howmuch+J)*degrees))*500,COS(RAD((howmuch+J)*degrees))*500
190 PLOT85,SIN(RAD((howmuch+J+1)*degrees))*500,COS(RAD((howmuch+J+1)*degrees))*500
200 NEXT
210 GCOLOR,3:MOVE0,0
220 DRAW SIN(RAD(howmuch*degrees))*500,COS(RAD(howmuch*degrees))*500
230 MOVE0,0:DRAWSIN(RAD((howmuch+item_value(I))*degrees))*500,COS(RAD((howmuch+item_value(I))*degrees))*500
240 FOR K=1 TO LEN(item_name$(I))
250 VDU5:MOVE SIN(RAD((howmuch+item_value(I)/2)*degrees))*500,COS(RAD((howmuch+item_value(I)/2)*degrees))*500
260 GCOLOR,0:PRINTMID$(item_name$(I),K,1):NEXT:VDU4
270 howmuch=howmuch+item_value(I)
280 NEXT
290 GCOLOR,3:MOVE0,0
300 FORI=0 TO 360 STEP 360/total
310 DRAW SIN(RAD(I))*500,COS(RAD(I))*500
00
320 NEXT
330 ENDPROC
340 DEFPROCcircle
350 FOR I=1 TO item_number
360 total=total+item_value(I)
370 NEXT
380 degrees=360/total
390 ENDPROC
400 DEFPROCdata
410 PRINTTAB(5);"DATA ENTRY"
420 PRINT' " Enter item name"
430 PRINT' " and value"'
440 PRINT"(input ZZZ to end)"'
450 item_number=0

```

```

460 REPEAT
470 item_number=item_number+1
480 INPUT item_name$(item_number)
490 IF item_name$(item_number)="ZZZ" T
HEN 530
500 INPUT item_value(item_number)
510 PRINT
520 UNTIL FALSE
530 item_name$(item_number)="END OF FI
LE"
535 item_value(item_number)=999
537 item_number=item_number-1
540 CLS
550 ENDPROC

```

In this case PROCdata allows the user to input the data (this procedure could be adapted for use on HISTOGRAM). The item name and value are entered. The values do not have to add up to any particular total, PROCcircle works out the correct angle for each section.

PROCpiechart draws the piechart. Line 110 sets the graphics origin to the centre of the screen. There are two nested loops. The program goes round the outer loop (lines 130 to 280) once for every item in the data array. The program goes around the minor loop (lines 160 to 200) according to the value set against each item in the data. If the values are small then there will still be a complete pie, but the sides will be straight! Lines 170 and 180 move to the cursor through the graphics origin (now in the centre of the screen) and the previous point plotted on the outside of the pie. Line 190 calculates the next point around the pie and

PLOT 85, (horizontal position) (vertical position)

draws and fills a triangle between these three points.

Lines 210 to 230 draw a white line between each section of the pie, lines 290 to 320 draw a white circle around the outside. These are not strictly necessary but make the final result look much neater.

Lines 240 to 280 print the names given to each section onto the chart. This is another example of how VDU5 and VDU4 can be

used. You may not like the way that some of the titles are printed from top to bottom, in which case you can modify these lines yourself.

The final program was written because we wanted to write another program which included a world map, drawn by high resolution graphics. In order to collect all the necessary **data** we hit on the idea of writing a program which would let us move a cursor around the screen (perhaps underneath a tracing paper outline) to work out the necessary co-ordinates to plot a hi-res diagram. Here is that program. (By the way, we still haven't got around to writing the program with the world map!)

```

10 REM ***** PLOTTER *****
20 REM *** BY STEPHEN MILAN ***
30 MODE4
40 PROCinstructions
50 X=0:Y=0:action$="move"

60 DIM X1(100),Y1(100)
70 X1(1)=0:Y1(1)=0:oldX=0:oldY=0
80 REPEAT
90 IF INKEY(-33) THEN action$="move"
100 IF INKEY(-114) THEN action$="draw"
140 PROCmove
150 PRINTTAB(12,29);"X ";X;" ";TAB(20,29)
"Y ";Y;" "
160 PROCcross_hairs:IF action$="draw" THE
N FORI=1 TO 50:NEXT
170 PROCcross_hairs
180 IF INKEY(-106) AND action$="move" THE
N PLOT69,X,Y:oldX=X:oldY=Y:PROCco_Ord
190 IF INKEY(-106) AND action$="draw" THE
N MOVE oldX,oldY:DRAW X,Y:oldX=X:oldY=Y:PRO
Cco_Ord
200 UNTIL FALSE
210 DEF PROCcross_hairs
220 IF action$="draw" THEN 260
230 MOVE X-50,Y:PLOT 6,X-10,Y:MOVE X+10,Y
:PLOT 6,X+50,Y

```

```

240 MOVE X,Y-50:PLOT 6,X,Y-10:MOVE X,Y+10
:PLOT 6,X,Y+50
250 ENDPROC
260 MOVE oldX,oldY:PLOT 6,X,Y
270 ENDPROC
280 DEF PROCmove
290 IF INKEY(-58) THEN Y=Y+20
300 IF INKEY(-42) THEN Y=Y-20
310 IF INKEY(-26) THEN X=X-20
320 IF INKEY(-122) THEN X=X+20
330 IF X>1279 THEN X=X-20
340 IF X<0 THEN X=0
350 IF Y>1023 THEN Y=Y-20
360 IF Y<0 THEN Y=0
370 ENDPROC
380 DEF PROCcoLond
400 PRINT;action$;" : ";
410 PRINTTAB(12);"X ";X;" ";TAB(20)"Y ";
Y;" "
420 FOR I=1 TO 2000:NEXT:ENDPROC
450 DEF PROCInstructions
460 PRINTTAB(16,2);"PLOTTER"
470 PRINT'"This Program helps you to work
out the necessary Points for including hi-
-res drawings into your own Programs."
480 PRINT'"The Position of the cursor is
controlled by the cursor-control keys."
490 PRINT'"A Point is fixed by Pressing <
COPY>"
500 PRINT'"The functions available are :"'
510 PRINT'"      f0 - MOVE"
520 PRINT'"      f1 - DRAW"
560 PRINT'"TAB(8)"PRESS ANY KEY TO CONTIN
UE"
570 continue$=GET$
580 CLS
590 ENDPROC

```

The program goes around the REPEAT...UNTIL FALSE loop (lines 80 to 200). Lines 90 and 100 show a new way of using

INKEY. Any key when pressed will return a different value to INKEY. In lines 90 and 100 we want to test for red keys f0 and f1. If f0 is pressed it will return the value -33. If f1 is pressed then the value will be -114. There is a list of the values of all the keys in the User Guide. So if either of these keys is pressed then action\$ becomes either "move" or "draw". If any other key is pressed then action\$ is not affected.

PROCmove (lines 280 to 370) alters the co-ordinates X and Y. Again INKEY is used, this time to detect the four cursor control keys.

Line 150 prints the co-ordinate values onto row 29 of the screen. PROCcross-hairs (lines 210 to 250) shows the co-ordinates' position on the screen. Line 220 tests to see whether action\$ is "move" or "draw". If action\$ = "move", then lines 230 and 240 draw cross-hairs onto the screen, centred on the co-ordinates. These are drawn with PLOT 6 which is the "opposite colour" plot demonstrated earlier. This means that the cross-hairs can be erased without affecting what was on the screen before, just by calling PROCcross-hairs again (line 170).

If action\$="draw" then line 260 MOVEs the cursor to oldX, oldY (which is the last point fixed) and then draws PLOT6,X,Y. This then draws a line from the previous point fixed to the position X,Y. Because PLOT 6 is used, this line can be continually erased and the line moved until the required position is reached. This is known as "rubber-banding", and is used in many professional computer-aided design programs.

Lines 180 and 190 look for the COPY key to be pressed. This fixes the point. If action\$="move" then PLOT 69,X,Y draws a small dot at the position of the co-ordinates. If action\$="draw" then a line is drawn from oldX, oldY to X,Y. Then oldX is made equal to X, and oldY equal to Y so that these can be used as references if needed for drawing to the next co-ordinates.

PROCCo-ord prints the co-ordinates which have been fixed, at the bottom of the screen, so that these can be accurately noted down. If you need to be reminded to write them down add this line:

If you have a printer and want a printed list then add the lines:

```
395 VDU2
```

```
415 VDU3
```

The next section extends the program so that it can plot circles.

Add these lines:

```
110 IF INKEY(-115) THEN PROCcircle
530 PRINT' "      f2 ~ CIRCLE"
600 DEF PROCcircle
610 PRINTTAB(0,29);SPC(39);TAB(0,30);SPC(
39);
620 PRINTTAB(0,29);"Circle drawing routine.
Radius ?"
630 X=oldX:Y=oldY
640 REPEAT
650 PROCmove
660 PROCcross_hairs:PROCcross_hairs
670 IF INKEY(-21) THEN PROChome
680 UNTIL INKEY(-106)
690 XX=ABS(oldX-X):YY=ABS(oldY-Y)
700 radius=SQR(XX*XX+YY*YY)
710 PRINTTAB(0,30);"Plot circle radius ";
radius
720 Plot=4:FOR angle=0 TO 360 STEP 5
730 PLOT Plot,SIN(RAD(angle))*radius+oldX
,COS(RAD(angle))*radius+oldY
740 Plot=5
750 NEXT
760 X=oldX:Y=oldY
770 PRINTTAB(0,29);SPC(39)
780 ENDPROC
```

Line 110 detects red key f2.

PROCcircle first of all uses the cross-hairs to find the radius of the circle, which will be centred on the last fixed point. COPY fixes the radius, line 700 calculates the radius and the circle is drawn by lines 720 to 750.

The next addition allows you to print messages onto your display.

```

50 X=0:Y=0:action$="move":written=0
130 IF INKEY(-116) THEN PROCletters
390 IF written=1 THEN 430
430 PRINT"VDU5";TAB(12);message$;
440 ENDPROC
540 PRINT/"      f3 - MESSAGE"
790 DEF PROCletters
800 PROCprepape
810 REPEAT
820 letter$=GET$
830 IF (LEN(message$)+3)*32+oldX>1280 OR
LEN(message$)+3>20 THEN VDU7
840 IF (LEN(message$)+1)*32+oldX>1280 OR
LEN(message$)+1>20 THEN VDU7:PROCprepape
850 flag = ASC(letter$):IF flag=13 THEN 8
80
860 PRINTletter$;
870 message$=message$+letter$
880 UNTIL flag=13
890 VDU5
900 MOVE oldX,oldY
910 PRINT message$
920 VDU4
930 PRINTTAB(0,29);SPC(39)
940 PRINTTAB(0,30);SPC(39);CHR$(11)
950 written=1:PROCcoLond:written=0
960 ENDPROC
970 DEF PROCprepape
980 PRINTTAB(0,29);SPC(39)
990 PRINTTAB(0,30);SPC(39)
1000 PRINTTAB(0,29);"MESSAGE - input messa
ge <RETURN>"
1010 message$="":letter$="":%FX15,1
1020 ENDPROC

```

The variable **written** is a flag to make PROCco-ord PRINT the correct message.

The message is entered through the REPEAT. . .UNTIL loop (lines 810 to 880). This GETs each letter and checks that the message is not too long to fit into the space available (or just too long). There is another flag (called **flag**) in line 850 which looks for the **RETURN** key to be pressed. This lets the program escape from the loop and print the message at the correct place on the screen.

The last section will only work on a Model B machine, or a Model A with 32K of memory. This is because of the space needed to set up two arrays. The program stores all the co-ordinates that have been fixed. Then at any time, by pressing red key f4, the co-ordinates will jump to the nearest point previously fixed.

```

50 X=0:Y=0:action$="move":Points=1:writ
en=0
60 DIM X1(100),Y1(100)
70 X1(1)=0:Y1(1)=0:oldX=0:oldY=0
120 IF INKEY(-21) THEN PROChome
180 IF INKEY(-106) AND action$="move" THE
N PLOT69,X,Y:PROCarray:oldX=X:oldY=Y:PROCco
_Ord
190 IF INKEY(-106) AND action$="draw" THE
N MOVE oldX,oldY:DRAW X,Y:PROCarray:oldX=X:
oldY=Y:PROCco_Ord
550 PRINT' "      f4 - FIND NEAREST POINT"
1030 DEF PROChome
1040 closest=9999:FORI=1 TO Points
1050 distx=X1(I)-X:disty=Y1(I)-Y
1060 IF ABS(distx)+ABS(disty)<closest THEN
closest=ABS(distx)+ABS(disty):closestPoint
=I
1070 NEXT
1080 X=X1(closestPoint):Y=Y1(closestPoint)

1090 ENDPROC
1100 DEF PROCarray
1110 Points=Points+1
1120 X1(Points)=oldX:Y1(Points)=oldY
1130 ENDPROC

```

Line 60 DIMensions the two arrays X1 and Y1. Lines 180 and 190 have PROCarray added which adds any new co-ordinator to the list. PROChome (lines 1030 to 1090) searches through the arrays and finds the closest to the present co-ordinates. ABS (absolute) is a BASIC word which tells the computer to ignore + and - signs.

If you have a Model B and joysticks, these lines will let you move the cursor with one of the joysticks. Don't add the lines to the program unless you *are* going to add the joysticks, or nothing will work.

```
180 IF (INKEY(-106) OR (ADVAL(0) AND 3
)=1) AND action$="move" THEN PLOT69,X,Y:
PROCarray:oldX=X:oldY=Y:PROCcoLond
190 IF (INKEY(-106) OR (ADVAL(0) AND 3
)=1) AND action$="draw" THEN MOVE oldX,o
ldY:DRAW X,Y:PROCarray:oldX=X:oldY=Y:PRO
CcoLond
290 DX=INT((ADVAL(1)-30000)/2000):IF D
X<5 AND DX>-5 THEN DX=0
300 DY=INT((ADVAL(2)-30000)/2000):IF D
Y<5 AND DY>-5 THEN DY=0
310 X=X+DX
320 Y=Y-DY
680 UNTIL INKEY(-106) OR (ADVAL(0) AND
3)
```

WORD SQUARE 3

This wordsquare contains 44 words, most of which were introduced in Chapters 9-12.

S	C	T	D	E	P	L	O	G	I	C	A	L	H	E	N
M	H	F	O	R	E	G	R	O	U	N	D	I	L	A	N
C	A	U	U	S	T	H	E	Y	A	C	E	D	P	H	P
O	N	N	B	A	C	K	G	R	O	U	N	D	E	E	O
O	N	C	L	R	O	N	E	E	P	L	V	L	D	N	L
R	E	T	E	E	L	I	O	T	I	T	E	T	U	O	Y
D	L	I	H	L	O	T	X	E	T	E	L	E	T	I	G
I	R	O	E	E	U	I	V	M	C	V	O	C	I	T	O
N	H	N	I	A	R	A	M	A	H	O	P	A	L	A	N
A	R	K	G	S	S	W	A	R	D	N	E	Y	P	R	D
T	S	E	H	E	H	C	O	A	T	T	R	G	M	U	P
E	T	Y	T	O	L	I	K	P	E	O	L	O	A	D	T
S	E	R	G	C	O	L	R	A	M	U	D	V	G	H	L
E	C	H	A	R	A	C	T	E	R	E	R	Y	E	E	G
O	R	E	S	L	A	F	M	E	S	C	E	A	N	N	Y
A	C	T	U	A	L	P	T	M	X	K	A	T	T	D	T
H	Y	E	F	L	A	S	H	W	N	T	H	O	A	L	T
L	A	T	T	A	C	K	L	I	O	Y	D	E	R	U	D
D	N	I	A	T	S	U	S	O	C	C	T	O	L	P	K
S	O	R	I	G	I	N	Q	A	N	S	O	U	N	D	T

ACTUAL

AMPLITUDE

ATTACK

BACKGROUND

CHANNEL

CHARACTER

DRAW

DURATION

ENVELOPE

FALSE

FLASH

FOREGROUND

LOAD

LOGICAL

MAGENTA

MEMORY

MODE

MOVE

CHR

COLOUR

COORDINATES

CYAN

DECAY

DOUBLEHEIGHT

FUNCTIONKEY

GCOL

GRAPHICS

HIRES

INKEY

KILOBYTE

ORIGIN

PARAMETER

PITCH

PLOT

POLYGON

RAM

RELEASE

SAVE

SOUND

STEADY

SUSTAIN

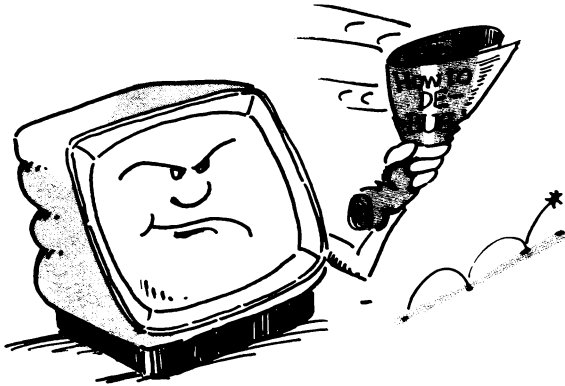
TELETEXT

TEXT

VDU

Appendix 1

DEBUGGING



What happens if the program doesn't work? Then it has a **bug** in it and needs **debugging**.

There are two main sorts of problems:

Errors of **syntax**

Errors of **logic**

SYNTAX ERRORS

If you are copying a program printed in a book or magazine then it's usually (although not always) an error of **syntax** which is the problem. These are the easiest to trace.

Syntax means "the rules of a language". A syntax error means that you are not obeying the rules of BASIC – you're not talking "proper" to the computer. As mentioned in Chapter 1, the compu-

ter is essentially very stupid and unless it receives exactly the right instructions it cannot do what you want it to. So any error made whilst the program was keyed-in can have a serious effect on the running.

If you have a syntax error then usually there will be some sort of **error message** printed on the screen together with a line number. This is very helpful. LIST that line and check it very carefully with the line printed in the book (or magazine or wherever you got the program from). Here are some of the most common errors.

Using small letters instead of CAPITAL LETTERS

The BBC computer considers a small **a** to be a completely different letter to a capital **A**, and so won't recognise any letters that are wrong in this way. You will need to copy any lines where this has happened, substituting the correct characters.

Using 1 (one) instead of I

The letter **I** is very popular with computer programmers for use as a variable name. This is because **I** is the closest key to (and). In programs using arrays it is very quick to key in **A(I)**. So you will find a lot of programs which use **I** for loop parameters in arrays. Be careful not to confuse these with the number one (1).

Using "O" instead of 0

Most program listings show zero as Ø. The BBC micro in **MODE 7** uses teletext characters where the zero has no line but is more "diamond" shaped than the letter "O". Be careful not to mix them up.

Using a decimal point (.) instead of a comma (,)

Easy to do if your finger slips, but quite difficult to trace, because you're not likely to get an error message, particularly if the bug is in a **DATA** statement.

No RETURN between statements

That is where you copy two statement lines together and don't

press RETURN after the first one. The easiest way to spot this is to LIST just one line. If two lines appear together then you haven't pressed RETURN between them. The remedy is simple. Move the cursor up, copy the first line and then press RETURN. Then move the cursor up again and copy the second line.

Wrong spaces

Spaces can cause trouble in some cases, either because there are too many, or not enough. For example, there should never be a space between TAB and the bracket:

TAB(10) *not* TAB (10)

There *must* be a space between a variable name and a following BASIC word. Thus:

```
70 IF lines = number AND rows = total THEN flag=1
not 70 IF lines = numberAND rows = totalTHEN flag=1
```

After that, the chances are that you've misspelt something, so you'll need to keep looking and checking. It's always better if you can get someone to read out the program while you type or check. Often you can stare at a bug for ages before you actually see it, so it's a good help to get someone else to look, for sometimes they see it straight away.

If you have used the LISTO command to introduce spacing into a list to make it easier to read, and then use the COPY key to make a correction, the extra spaces will be copied into the new line, whether you want them or not. If you have used the WIDTH command and then COPY, you may get spaces introduced into the middle of lines, perhaps in the middle of a variable name, or a BASIC keyword or a number. So if you have been using LISTO and are going to use COPY, then it's best to type LISTOØ (and if you have been using WIDTH, WIDTH79).

Spaces in DATA statements are very important. Spaces between a DATA item and the comma, or an extra space at the end of a DATA line before RETURN is pressed, will be counted as part of the item. A comma at the end of a DATA line will be understood by the computer as an extra (blank) item.

LOGICAL ERRORS

An error of **logic** means that you have (correctly) told the computer to do something that won't work. Your **algorithm** is wrong. Usually the computer won't give an error message because it cannot tell that anything is wrong. If you are writing a program to perform a calculation, it is best to test it with data where you know the correct answer.



Cheerful Thought –

It is only possible to *prove* that a program is wrong. You can't *prove* that it's correct.

Index

ABSolute	157
Actual colour	119
ADVAL	103, 157
Amplitude	128
Arrays	55
Auto line numbering	55
Auto repeat	86
BASIC	10
Brackets	31
Break	22
Bugs	16
Caps lock	38
CHAIN	26
CHANNEL	128
CHR\$	95
CLS	49
COLOUR	117
Connecting up	13
CONTROL-L	21
CONTROL-N	67
Co-ordinates	140
COPY	21
Cursor control keys	20
DATA	60
DEF PROC	66

DELETE	15
DIMension	56
Direct commands	17, 29
DRAW	140
Duration	129
Editing	19
Equals	23, 37
ELSE	44
END	15
ENDPROC	66
ENVELOPE	131
Errors	50, 159
Escape	55
EVAL	125
Flash	109
Floating point variable	52
Flowchart	39
FOR. . .NEXT	47
GCOL	144
GET\$	51
GOSUB	74
GOTO	41
Graphics	134, 139
Graphics origin	139
Greater than (>)	37
IF. . .THEN	41
IF. . .THEN. . .ELSE	44
INKEY\$	104, 128
INPUT	15, 24
INTEger	34
Integer variables	52
Joysticks	103, 157
LEFT\$	78
LEN	76

Less than (<)	37
LIST	17
LISTO	48
LOAD	26
Loading	25
Logical colours	119
Logical errors	162
Loop parameter	49
Memory	24
MID\$	78
Missing	31
Mistake	16
MOVE	139
MODE	93
Multi-statement lines	65
Musical notes	129
NEW	22
NEXT	47
Not equal to (<>)	37
OLD	23
Origin	139
Output	24
PI	34
Pitch	128
PLOT	142
PRINT	15
PROCedure	63
Process	24
Program	10
READ	61
REM	58
REPEAT. . .UNTIL	51
RESTORE	62
RETURN key	15
RETURN	74

RIGHT\$	81
RND	51, 53
RUN	16
SAVE	27
Saving	25
Sorting	58
SOUND	127
Speeding up programs	52
SQR (square root)	32
Square	31
STEP	48
String variables	19
Structured programming	73
Subroutines	73
Subscripted variables	56
Syntax errors	16, 159
TAB	75
Teletext colours	108
Text origin	139
TIME	82
UNTIL	51
User defined characters	96
User defined keys	104
Variable names	18
VDU 19	119
VDU 23	96

Answers to questions on page 99.

10 VDU 23, 224, 32, 36, 66, 129, 66, 36, 32

20 VDU 23, 225, 32, 32, 68, 68, 136, 136, 255, 255

30 VDU 23, 226, 20, 60, 124, 214, 254, 214, 254, 254

A YOUNG PERSON'S GUIDE TO BBC BASIC

The personal computer is the modern equivalent of the genie in the bottle. Correctly instructed, it can transport the user to realms limited only by human imagination. The computer can be helper, teacher, companion... depending on the programs used.

BBC BASIC, developed for the BBC Computer, is a version of the computer language BASIC. It quickly became recognised as one of the fastest and most powerful versions of BASIC available on a personal computer, and is now available on two other Acorn computers: the Electron and the Atom.

This book has been written with the young first-time user in mind, but it will also appeal to older and more experienced users. It is especially aimed at those people interested in writing their own programs. It assumes no previous knowledge of computing, and pays particular attention to good programming technique.

Michael Milan is a television producer specialising in developing training programmes on computer subjects. As a member of NCC's Training Development team, he has worked on all their recent training packages. His two sons – Stephen (14) and Christopher (11) – helped to write and test the programs used in the present book.